PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04B | A2

(11) International Publication Number: WO 98/26510

(43) International Publication Date: 18 June 1998 (18.06.98)

(21) International Application Number: PCT/US97/22550

(22) International Filing Date: 8 December 1997 (08.12.97)

(30) Priority Data:
60/032,485 9 December 1996 (09.12.96) US

(71) Applicant: PACKETEER, INC. [US/US]; Suite 305, 307 Orchard City Drive, Campbell, CA 95008 (US).

(72) Inventor: PACKER, Robert, L.; 16095 Redwood Lodge Road, Los Gatos, CA 95036 (US).

(74) Agents: ALLEN, Kenneth, R. et al.; Townsend and Townsend and Crew LLP, Two Embarcadero Center, 8th floor, San Francisco, CA 94111-3834 (US).

(81) Designated States: AU, CN, IL, JP, KR, NZ.

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: METHOD FOR MANAGING FLOW BANDWIDTH UTILIZATION AT NETWORK, TRANSPORT AND APPLICATION LAYERS IN STORE AND FORWARD NETWORK

(57) Abstract

In a packet communication environment, a method is provided for classifying packet network flows for use in determining a policy, or rule of assignment of a service level, and enforcing that policy by direct rate control. The method comprises applying individual instances of traffic objects, i.e., packet network flows to a classification model based on selectable information obtained from a plurality of layers of a multi-layered communication protocol, then mapping the flow to the defined traffic classes, which are arbitrarily assignable by an offline manager which creates the classification. It is useful to note that the classification need not be a complete enumeration of the possible traffic.

METHOD FOR MANAGING FLOW BANDWIDTH UTILIZATION AT NETWORK, TRANSPORT AND APPLICATION LAYERS IN STORE AND FORWARD NETWORK

COPYRIGHT NOTICE

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority from the following U.S. Provisional Application:

U.S. Provisional Patent Application Serial No. 60/032,485, Robert L. Packer, entitled, "Method for Managing Flow Bandwidth Utilization at Network, Transport and Application Layers in Store and Forward Network", filed December 9, 1996.

Further, this application makes reference to the following commonly owned U.S. Patent Application:

Copending U.S. Patent Application Serial No. 08/762,828, in the name of Robert L. Packer, entitled "Method for Rapid Data Rate Detection in a Packet Communication Environment Without Data Rate Supervision," relates to a technique for automatically determining the data rate of a TCP connection.

Further, this application makes reference to the following U.S. Patent Application:

Copending U.S. Patent Application Serial No. 08/742,994, in the name of Robert L. Packer, entitled "Method for Explicit Data Rate Control in a Packet Communication Environment Without a Data Rate Supervision," relates to a technique for automatically scheduling TCP packets for transmission.

BACKGROUND OF THE INVENTION

This invention relates to digital packet telecommunications, and particularly to management of network bandwidth based on information ascertainable from multiple layers of OSI network model. It is particularly useful in conjunction with data flow rate detection and control of a digitally-switched packet telecommunications environment normally not subject to data flow rate control.

The ubiquitous TCP/IP protocol suite, which implements the world-wide data communication network environment called the Internet and is also used in private networks (Intranets), intentionally omits explicit supervisory function over the rate of data transport over the various media which comprise the network. While there are certain perceived advantages, this characteristic has the consequence of juxtaposing very high-speed packet flows and very low-speed packet flows in potential conflict for network resources, which results in inefficiencies. Certain pathological loading conditions can result in instability, overloading and data transfer stoppage. Therefore, it is desirable to provide some mechanism to optimize efficiency of data transfer while minimizing the risk of data loss. Early indication of the rate of data flow which can or must be supported is very useful. In fact, data flow rate capacity information is a key factor for use in resource allocation decisions.

Internet/Intranet technology is based largely on the TCP/IP protocol suite, where IP, or Internet Protocol, is the <u>network layer</u> protocol and TCP, or Transmission Control Protocol, is the <u>transport layer</u> protocol. At the network level, IP provides a "datagram" delivery service. By contrast, TCP builds a transport level service over the datagram service to provide guaranteed, sequential delivery of a byte stream between two IP hosts.

TCP flow control mechanisms operate exclusively at the end stations to limit the rate at which TCP endpoints emit data. However, TCP lacks explicit data rate control. In fact, there is heretofore no concept of coordination of data rates among multiple flows. The basic TCP flow control mechanism is a sliding window, superimposed on a range of bytes beyond the last explicitly-acknowledged byte. Its sliding operation limits the amount of unacknowledged transmissible data that a TCP endpoint can emit.

Another flow control mechanism is a congestion window, which is a refinement of the sliding window scheme, which employs conservative expansion to fully

utilize all of the allowable window. A component of this mechanism is sometimes referred
to as "slow start".

The sliding window flow control mechanism works in conjunction with the Retransmit Timeout Mechanism (RTO), which is a timeout to prompt a retransmission of unacknowledged data. The timeout length is based on a running average of the Round Trip Time (RTT) for acknowledgment receipt, i.e. if an acknowledgment is not received within (typically) the smoothed RTT + 4*mean deviation, then packet loss is inferred and the data pending acknowledgment is retransmitted.

Data rate flow control mechanisms which are operative end-to-end without explicit data rate control draw a strong inference of congestion from packet loss (inferred, typically, by RTO). TCP end systems, for example, will 'back-off', i.e., inhibit transmission in increasing multiples of the base RTT average as a reaction to consecutive packet loss.

Bandwidth Management in TCP/IP Networks

Conventional bandwidth management in TCP/IP networks is accomplished by a combination of TCP end systems and routers which queue packets and discard packets when certain congestion thresholds are exceeded. The discarded, and therefore unacknowledged, packet serves as a feedback mechanism to the TCP transmitter. (TCP end systems are clients or servers running the TCP transport protocol, typically as part of their operating system.)

The term "bandwidth management" is often used to refer to link level bandwidth management, e.g. multiple line support for Point to Point Protocol (PPP). Link level bandwidth management is essentially the process of keeping track of all traffic and deciding whether an additional dial line or ISDN channel should be opened or an extraneous one closed. The field of this invention is concerned with *network* level bandwidth management, i.e. policies to assign available bandwidth from a single logical link to network flows.

Routers support various queuing options. These options are generally intended to promote fairness and to provide a rough ability to partition and prioritize separate classes of traffic. Configuring these queuing options with any precision or without side effects is in fact very difficult, and in some cases, not possible. Seemingly simple things, such as the length of the queue, have a profound effect on traffic characteristics. Discarding packets as a feedback mechanism to TCP end systems may cause large, uneven delays perceptible to interactive users.

In a copending U.S. Patent Application Serial No. 08/742,994, in the name of Robert L. Packer, entitled "Method for Explicit Data Rate Control in a Packet Communication Environment Without Data Rate Supervision," a technique for automatically scheduling TCP packets for transmission is disclosed. Furthermore, in a copending U.S. Patent Application Serial No. 08/762,828, in the name of Robert L. Packer, entitled "Method for Rapid Data Rate Detection in a Packet Communication Environment Without Data Rate Supervision," a technique for automatically determining the data rate of a TCP connection is disclosed. While these patent applications teach methods for solving problems associated with scheduling transmissions and for automatically determining a data flow rate on a TCP connection, respectively, there is no teaching in the prior art of methods for explicitly managing TCP packet traffic based upon information about the flow's characteristics at multiple OSI protocol layers.

Bandwidth management is heretofore not known to employ information contained in the packets corresponding to higher OSI protocol layers, even though such information may be extremely useful in making bandwidth allocation and management decisions.

SUMMARY OF THE INVENTION

According to the invention, in a packet communication environment, a method is provided for classifying packet network flows for use in determining a policy, or rule of assignment of a service level, and enforcing that policy by direct rate control. The method comprises applying individual instances of traffic objects, i.e., packet network flows to a classification model based on selectable information obtained from a plurality of layers of a multi-layered communication protocol, then mapping the flow to the defined traffic classes, which are arbitrarily assignable by an offline manager which creates the classification. It is useful to note that the classification need not be a complete enumeration of the possible traffic.

In one aspect of the invention, bandwidth may be divided into arbitrary units, partitions, facilitating isolation and allocation. A partition is allocated for a class or set of traffic classes, carving the bandwidth of the associated link into multiple, independent pieces.

In another aspect of the invention, available bandwidth may be allocated among flows according to a policy, which may include any combination of guaranteed information rate, excess infonnation rate, the later allocated according to a priority.

In another aspect of the invention, bandwidth resource needs of multiple heterogeneous requesting flows are reconciled with available bandwidth resources in accordance with policy of each flow based upon the flow's class. Flows requiring reserved service with guaranteed information rates, excess information rates or unreserved service are reconciled with the available bandwidth resources continuously and automatically.

In another aspect of the invention, providing an admissions policy which is invoked whenever a request for a bandwidth cannot be met consistently with other users of bandwidth.

An advantage of network management techniques according to the present invention is that network managers need only define traffic classes which are of interest.

A further advantage of the present invention is that traffic classes may include information such as a URI for web traffic.

A yet further advantage of the present invention is that service levels may be defined in terms of explicit rates and may be scaled to a remote client or server's network access rate. Different service levels may be specified for high speed and low speed users.

A yet further advantage of the present invention is that service levels may be defined in terms of a guaranteed minimum service level.

The invention will be better understood upon reference to the following detailed description in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1A depicts a representative client server relationship in accordance with a particular embodiment of the invention;

Fig. 1B depicts a functional perspective of the representative client server relationship in accordance with a particular embodiment of the invention;

Fig. 1C depicts a representative internetworking environment in accordance with a particular embodiment of the invention;

Fig. 1D depicts a relationship diagram of the layers of the TCP/IP protocol suite;

Fig. 1E depicts a two dimensional representation of timing relationships in the exchange of packets between hosts using the TCP protocol;

Figs. 2A-2B depict representative divisions of bandwidth according to a particular embodiment of the invention;

Figs. 2C-2E are flow charts depicting process steps according to a particular embodiment of the invention;

Fig. 3 is a block diagram of a particular embodiment according to the invention;

Fig. 4A is a block diagram of a data structure according to a particular embodiment of the invention;

Fig. 4B is a block diagram of data structure according to a particular embodiment of the invention; and

Figs. 5A-5H are flow charts depicting process steps according to a particular embodiment of the invention.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

A preferable embodiment of a flow bandwidth management system according to the invention has been reduced to practice and will be made available under the trade name "PacketShaper™."

1.0 Introduction

The present invention provides techniques to manage network bandwidth such as on a network access link between a local area network and a wide area network. Systems according to the present invention enable network managers to: define traffic classes; create policies which define service levels for traffic classes; and isolate bandwidth resources associated with certain traffic classes. Inbound as well as outbound traffic may be managed. Table 1 provides a definitional list of terminology used herein.

LIST OF DEFINITIONAL TERMS

ADMISSIONS CONTROL	A policy invoked whenever a system according to the invention detects that a guaranteed information rate cannot be maintained. An admissions control policy is analogous to a busy signal in the telephone world.
CLASS SEARCH ORDER	A search method based upon traversal of a N-ary tree data structure containing classes.
COMMITTED INFORMATION RATE (CIR)	A rate of data flow allocated to reserved service traffic for rate based bandwidth allocation for a committed bandwidth. Also called a guaranteed information rate (GIR).
EXCEPTION	A class of traffic provided by the user which supersedes an automatically determined classification order.
EXCESS INFORMATION RATE (EIR)	A rate of data flow allocated to reserved service traffic for rate based bandwidth allocation for uncommitted bandwidth resources.
FLOW	A flow is a single instance of a traffic class. For example, all packets in a TCP connection belong to the same flow. As do all packets in a UDP session.
GUARANTEED INFORMATION RATE (GIR)	A rate of data flow allocated to reserved service traffic for rate based bandwidth allocation for a committed bandwidth. Also called a committed information rate (CIR).

HARD ISOLATION	Hard isolation results from the creation of an entirely separate logical channel for a designated set of classes.
INSIDE	On the system side of an access link. Outside clients and servers are on the other side of the access link.
ISOLATION	Isolation is the degree that bandwidth resources are allocable to traffic classes.
OUTSIDE	On the opposite side of an access link as viewed from the perspective of the system on which the software resides.
PARTITION	Partition is an arbitrary unit of network resources.
POLICY	A rule for the assignment of a service level to a flow.
POLICY INHERITANCE	A method for assigning policies to flows for which no policy exists in a hierarchical arrangement of policies. For example, if a flow is determined to be comprised of FTP packets for Host A, and no corresponding policy exists, a policy associated with a parent node, such as an FTP policy, may be located and used. See also POLICY SEARCH ORDER.
POLICY BASED SCALING	An adjustment of a requested data rate for a particular flow based upon the policy associated with the flow and information about the flow's potential rate.
RESERVED SERVICE	Reserved service is a service level intended for traffic which "bursts" or sends chunks of data. Reserved service is defined in terms of a scaled rate.

SCALED RATE	Assignment of a data rate based upon detected speed.
SERVICE LEVEL	A service paradigm having a combination of characteristics defined by a network manager to handle a particular class of traffic. Service levels may be designated as either reserved or unreserved.
SOFT ISOLATION	Restricting GIR allocated for traffic classes in a partition.
TARGET RATE	A target rate is a combination of a guaranteed rate and an excess rate. Target rate is a policy-based paradigm. Excess rate is allocated by systems according to the invention from bandwidth that is not consumed by reserved service. Policies will demand excess rate at a given priority and systems according to the invention satisfy this demand by a priority level.
TRAFFIC CLASS	All traffic between a client and a server endpoints. A single instance of a traffic class is called a flow. Traffic classes have properties or class attributes such as, *directionality*, which is the property of traffic to be flowing inbound or outbound.
UNRESERVED SERVICE	Unreserved service is a service level defined in terms of priority in which no reservation of bandwidth is made.

Table 1

1.1 Hardware Overview

The method for flow bandwidth management in a packet oriented telecommunications network environment of the present invention is implemented in the C programming language and is operational on a computer system such as shown in Fig. 1A. This invention may be implemented in a client-server environment, but a client-server environment is not essential. This figure shows a conventional client-server computer system which includes a server 20 and numerous clients, one of which is shown as client 25. The use of the term "server" is used in the context of the invention, wherein the server receives queries from (typically remote) clients, does substantially all the processing necessary to formulate responses to the queries, and provides these responses to the clients. However, server 20 may itself act in the capacity of a client when it accesses remote databases located at another node acting as a database server.

The hardware configurations are in general standard and will be described only briefly. In accordance with known practice, server 20 includes one or more processors 30 which communicate with a number of peripheral devices via a bus subsystem 32. These peripheral devices typically include a storage subsystem 35, comprised of a memory subsystem 35a and a file storage subsystem 35b holding computer programs (e.g., code or instructions) and data, a set of user interface input and output devices 37, and an interface to outside networks, which may employ Ethernet, Token Ring, ATM, IEEE 802.3, ITU X.25, Serial Link Internet Protocol (SLIP) or the public switched telephone network. This interface is shown schematically as a "Network Interface" block 40. It is coupled to corresponding interface devices in client computers via a network connection 45.

Client 25 has the same general configuration, although typically with less storage and processing capability. Thus, while the client computer could be a terminal or a low-end personal computer, the server computer is generally a high-end workstation or mainframe, such as a SUN SPARC server. Corresponding elements and subsystems in the client computer are shown with corresponding, but primed, reference numerals.

Bus subsystem 32 is shown schematically as a single bus, but a typical system has a number of buses such as a local bus and one or more expansion buses (e.g., ADB, SCSI, ISA, EISA, MCA, NuBus, or PCI), as well as serial and parallel ports. Network connections are usually established through a device such as a network adapter on one of these expansion buses or a modem on a serial port. The client computer may be a desktop system or a portable system.

The user interacts with the system using interface devices 37' (or devices 37 in a standalone system). For example, client queries are entered via a keyboard, communicated to client processor 30', and thence to modem or network interface 40' over bus subsystem 32'. The query is then communicated to server 20 via network connection 45. Similarly, results of the query are communicated from the server to the client via network connection 45 for output on one of devices 37' (say a display or a printer), or may be stored on storage subsystem 35'.

Fig. 1B is a functional diagram of a computer system such as that of Fig. 1A. Fig. 1B depicts a server 20, and a representative client 25 of a plurality of clients which may interact with the server 20 via the Internet 45 or any other communications method. Blocks to the right of the server are indicative of the processing steps and functions which occur in the server's program and data storage indicated by blocks 35a and 35b in Fig. 1A. A TCP/IP "stack" 44 works in conjunction with Operating System 42 to communicate with processes over a network or serial connection attaching Server 20 to Internet 45. Web server software 46 executes concurrently and cooperatively with other processes in server 20 to make data objects 50 and 51 available to requesting clients. A Common Gateway Interface (CGI) script 55 enables information from user clients to be acted upon by web server 46, or other processes within server 20. Responses to client queries may be returned to the clients in the form of a Hypertext Markup Language (HTML) document outputs which are then communicated via Internet 45 back to the user.

Client 25 in Fig. 1B possesses software implementing functional processes operatively disposed in its program and data storage as indicated by block 35a' in Fig. 1A. TCP/IP stack 44', works in conjunction with Operating System 42' to communicate with processes over a network or serial connection attaching Client 25 to Internet 45. Software implementing the function of a web browser 46' executes concurrently and cooperatively with other processes in client 25 to make requests of server 20 for data objects 50 and 51. The user of the client may interact via the web browser 46' to make such queries of the server 20 via Internet 45 and to view responses from the server 20 via Internet 45 on the web browser 46'.

1.2 <u>Network Overview</u>

Fig. 1C is illustrative of the internetworking of a plurality of clients such as client 25 of Figs. 1A and 1B and a plurality of servers such as server 20 of Figs. 1A and 1B as described herein above. In Fig. 1C, network 70 is an example of a Token Ring or frame oriented network. Network 70 links host 71, such as an IBM RS6000 RISC workstation, which may be running the AIX operating system, to host 72, which is a personal computer, which may be running Windows 95, IBM 0S/2 or a DOS operating system, and host 73, which may be an IBM AS/400 computer, which may be running the OS/400 operating system. Network 70 is internetworked to network 60 via a system gateway which is depicted here as router 75, but which may also be a gateway having a firewall or a network bridge. Network 60 is an example of an Ethernet network that interconnects host 61, which is a SPARC workstation, which may be running SUNOS operating system with host 62, which may be a Digital Equipment VAX6000 computer which may be running the VMS operating system.

Router 75 is a network access point (NAP) of network 70 and network 60. Router 75 employs a Token Ring adapter and Ethernet adapter. This enables router 75 to interface with the two heterogeneous networks. Router 75 is also aware of the Inter-network Protocols, such as ICMP ARP and RIP, which are described herein below.

Fig. 1D is illustrative of the constituents of the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite. The base layer of the TCP/IP protocol suite is the physical layer 80, which defines the mechanical, electrical, functional and procedural standards for the physical transmission of data over communications media, such as, for example, the network connection 45 of Fig. 1A. The physical layer may comprise electrical, mechanical or functional standards such as whether a network is packet switching or frame-switching; or whether a network is based on a Carrier Sense Multiple Access/Collision Detection (CSMA/CD) or a frame relay paradigm.

Overlying the physical layer is the data link layer 82. The data link layer provides the function and protocols to transfer data between network resources and to detect errors that may occur at the physical layer. Operating modes at the datalink layer comprise such standardized network topologies as IEEE 802.3 Ethernet, IEEE 802.5 Token Ring, ITU X.25, or serial (SLIP) protocols.

Network layer protocols 84 overlay the datalink layer and provide the means for establishing connections between networks. The standards of network layer protocols provide operational control procedures for internetworking communications and routing information through multiple heterogenous networks. Examples of network layer protocols are the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). The Address Resolution Protocol (ARP) is used to correlate an Internet address and a Media Access Address (MAC) for a particular host. The Routing Information Protocol (RIP) is a dynamic routing protocol for passing routing information between hosts on networks. The Internet Control Message Protocol (ICMP) is an internal protocol for passing control messages between hosts on various networks. ICMP messages provide feedback about events in the network environment or can help determine if a path exists to a particular host in the network environment. The latter is called a "Ping". The Internet Protocol (IP) provides the basic mechanism for routing packets of information in the Internet. IP is a non-reliable communication protocol. It provides a "best efforts" delivery service and does not commit network resources to a particular transaction, nor does it perform retransmissions or give acknowledgments.

The transport layer protocols 86 provide end-to-end transport services across multiple heterogenous networks. The User Datagram Protocol (UDP) provides a connectionless, datagram oriented service which provides a non-reliable delivery mechanism for streams of information. The Transmission Control Protocol (TCP) provides a reliable session-based service for delivery of sequenced packets of information across the Internet. TCP provides a connection oriented reliable mechanism for information delivery.

The session, or application layer 88 provides a list of network applications and utilities, a few of which are illustrated here. For example, File Transfer Protocol (FTP) is a standard TCP/IP protocol for transferring files from one machine to another. FTP clients establish sessions through TCP connections with FTP servers in order to obtain files. Telnet is a standard TCP/IP protocol for remote terminal connection. A Telnet client acts as a terminal emulator and establishes a connection using TCP as the transport mechanism with a Telnet server. The Simple Network Management Protocol (SNMP) is a standard for managing TCP/IP networks. SNMP tasks, called "agents", monitor network status parameters and transmit these status parameters to SNMP tasks called "managers." Managers track the status of associated networks. A Remote Procedure Call (RPC) is a programming interface which enables programs to invoke remote functions on server machines. The Hypertext Transfer Protocol (HTTP) facilitates the transfer of data objects across networks via a system of uniform resource indicators (URI).

The Hypertext Transfer Protocol is a simple protocol built on top of Transmission Control Protocol (TCP). It is the mechanism which underlies the function of the World Wide Web. The HTTP provides a method for users to obtain data objects from various hosts acting as servers on the Internet. User requests for data objects are made by means of an HTTP request, such as a GET request. A GET request as depicted below is comprised of 1) an HTTP protocol version, such as "http:/1.0"; followed by 2) the full path of the data object; followed by 3) the name of the data object. In the GET request shown below, a request is being made for the data object with a path name of "/pub/" and a name of "MyData.html":

HTTP-Version GET /pub/MyData.html (1)

Processing of a GET request entails the establishing of an TCP/IP connection with the server named in the GET request and receipt from the server of the data object specified. After receiving and interpreting a request message, a server responds in the form of an HTTP RESPONSE message.

Response messages begin with a status line comprising a protocol version followed by a numeric Status Code and an associated textual Reason Phrase. These elements are separated by space characters. The format of a status line is depicted in line (2):

Status-Line = HTTP-Version Status-Code Reason-Phrase (2)

The status line always begins with a protocol version and status code, e.g., "HTTP/1.0 200 ". The status code element is a three digit integer result code of the attempt to understand and satisfy a prior request message. The reason phrase is intended to give a short textual description of the status code.

The first digit of the status code defines the class of response. There are five categories for the first digit. 1XX is an information response. It is not currently used. 2XX is a successful response, indicating that the action was successfully received, understood and accepted. 3XX is a redirection response, indicating that further action must be taken in order to complete the request. 4XX is a client error response. This indicates a bad syntax in the request. Finally, 5XX is a server error. This indicates that the server failed to fulfill an apparently valid request.

Particular formats of HTTP messages are described in, Crocker, D., "Standard for the Format of ARPA Internet Text Messages", STD 11, RFC 822, UDEL, August 1982.

1.3 TCP Data Link Rate and Latency Period Determination

Techniques for determining data rates are more fully set forth in co-owned U.S. Patent Application Serial No. 08/762,828, entitled "Method for Rapid Data Rate Detection in a Packet Communication Environment Without Data Rate Supervision".

Fig. 1E depicts a two-dimensional timing diagram illustrative of one particular method for determining a data link rate and latency period from an exchange of packets between TCP endpoints. According to this method, the initial data packets are examined as they establish a connection. Parameters are developed from which round trip time (RTT) and maximum data rate can be determined. The serialization speed (SS) or data flow rate capacity of a link is given by the relation:

$$SS = m/T_1 \qquad (3)$$

where:

m = number of bytes in the first data packet (Data1)

T_1 = The arrival time of the first data packet less the time of arrival of the ACK packet (T_{data1} -T_{ACK})

Fig. 1E shows a two dimensional timing diagram of the connection protocol of a remote HTTP request between a local TCP endpoint (server) and a remote TCP endpoint (client). The remote endpoint issues a request for connection in the form of a SYN packet in step A. The SYN packet takes a finite but unknown transit time to arrive at the local TCP endpoint. The local TCP endpoint responds by sending its own SYN packet in step B. This SYN packet is of a known byte length and is issued at a known time, which becomes the reference time, "tbase." After a brief latency, the remote TCP endpoint issues a standard ACK packet, whose length is likewise known, in step C, and then also issues the first data packet data1 in step D.

Time T_1 is computed immediately at the time of arrival of data1 by examining the difference in arrival time of the received ACK packet and the data1 packet. The value of m is extracted by examining the entire first packet to determine its length or by examining the packet length field, if any, in the header information of the packet. A first estimate of serialization speed SS is given by Equation (3). Serialization speed can be estimated immediately upon interchange of the first packets and used to make important strategic decisions about the nature and speed of the connection about to be established.

An alternative method for determining a data link rate and latency period from an exchange of packets between TCP endpoints develops parameters from which round trip time (RTT) and maximum data rate can be determined. Values are obtained for serialization of n, the size (i.e., data length) of the SYN packet in response, plus the size (i.e., data length) of the ACK packet. Serialization time is determined according to equation (4) by dividing the value n by the transit time T_0, or

$$SS(\text{max possible}) = n/T_0 \qquad (4)$$

where

n =	number of bytes in the SYN packet plus the number of bytes in the ACK packet and
T_0 =	the arrival time of the ACK packet less the tbase value.

The round trip time minus the serialization time is the round trip latency T_d.

2.0 Traffic Class

A traffic class is broadly defined as traffic between one or more clients and one or more servers. A single instance of a traffic class is called a flow. Traffic classes have the property, or class attribute, of being directional, i.e. all traffic flowing inbound will belong to different traffic classes and be managed separately from traffic flowing outbound. The directional property enables asymmetric classification and control of traffic, i.e., inbound and outbound flows belong to different classes which may be managed independent of one another.

Traffic classes may be defined at any level of the TCP/IP protocol. For example, at the IP level, traffic may be defined as only those flows between a set of inside and outside IP addresses or domain names. An example of such a low level traffic class definition would be all traffic between my network and other corporate offices throughout the Internet. At the application level, traffic classes may be defined for specific URIs within a web server. Traffic classes may be defined having "Web aware" class attributes. For example, a traffic class could be created such as all URIs matching "*.html" for all servers, or all URIs matching "*.gif" for server X, or for access to server Y with URI "/sales/*" from client Z, wherein '*' is a wildcard character, i.e., a character which matches all other character combinations. Traffic class attributes left unspecified will simply match any value for that attribute. For example, a traffic class that accesses data objects within a certain directory path of a web server is specified by a URI of the directory path to be managed, e.g. "/sales/*".

2.1 Classifying Traffic

The present invention provides a method for classifying traffic according to a definable set of classification attributes selectable by the manager, including selecting a subset of traffic of interest to be classified. The invention provides the ability to classify and search traffic based upon multiple orthogonal classification attributes. Traffic class membership may be hierarchical. Thus, a flow may be classified by a series of steps through a traffic class tree, with the last step (i.e., at the leaves on the classification tree) mapping the flow to a policy. The policy is a rule of assignment for flows. For example, the first step in classification may be to classify a flow as web traffic, the next may further classify this flow as belonging to server X, and the final classification may be a policy for URI "*.avi".

A classification tree is a data structure representing the hierarchical aspect of traffic class relationships. Each node of the classification tree represents a class, and has a traffic specification, i.e., a set of attributes or characteristics describing the traffic, and a mask associated with it. Leaf nodes of the classification tree contain policies. According to a particular embodiment, the classification process checks at each level if the flow being classified matches the attributes of a given traffic class. If it does, processing continues down to the links associated with that node in the tree. If it does not, the class at the level that matches determines the policy for the flow being classified. If no policy specific match is found, the flow is assigned the default policy.

In a preferable embodiment, the classification tree is an N-ary tree with its nodes ordered by specificity. For example, in classifying a particular flow in a classification tree ordered first by organizational departments, the attributes of the flow are compared with the traffic specification in each successive department node and if no match is found, then processing proceeds to the next subsequent department node. If no match is found, then the final compare is a default "match all" category. If, however, a match is found, then classification moves to the children of this department node. The child nodes may be ordered by an orthogonal paradigm such as, for example, "service type." Matching proceeds according to the order of specificity in the child nodes. Processing proceeds in this manner, traversing downward and from left to right in the classification tree, searching the plurality of orthogonal paradigms. Key to implementing this a hierarchy is that the nodes are arranged in decreasing order of specificity. This permits search to find the most specific class for the traffic before more general.

Table 2 depicts components from which Traffic classes may be built. Note that the orientation of the server (inside or outside) is specified. And as noted above, any traffic class component may be unspecified, i.e. set to match any value.

Traffic Class Components

Client Side	Server Side
IP Address/Domain Name	IP Address/Domain Name
	TCP or UDP Service, e.g. WWW, FTP, RealAudio, etc.
	URI for Web Service, e.g. "*.html", "*.gif", "/sales/*", etc.

Table 2

Figs. 2A and 2B depict representative allocations of bandwidth made by a hypothetical network manager as an example. In Fig. 2A, the network manager has decided to divide her network resources first by allocating bandwidth between Departments A and B. Fig 2A shows the resulting classification tree, in which Department A bandwidth resources 202 and Department B bandwidth resources 204 each have their own nodes representing a specific traffic class for that department. Each traffic class may have a policy attribute associated with it. For example, in Fig. 2A, the Department A resources node 202 has the policy attribute Inside Host Subnet A associated with it. Next, the network manager has chosen to divide the bandwidth resources of Department A among two applications. She allocates an FTP traffic class 206 and a World Wide Web server traffic class 208. Each of these nodes may have a separate policy attribute associated with them. For example, in Fig. 2A, the FTP node 206 for has an attribute Outside port 20 associated with it. Similarly, the network manager has chosen to divide network bandwidth resources of Department B into an FTP server traffic class 210 and a World Wide Web server traffic class 212. Each may have their own respective policies.

Fig. 2B shows a second example, wherein the network manager has chosen to first divide network bandwidth resource between web traffic and TCP traffic. She creates three traffic nodes, a web traffic node 220, a TCP traffic node 224 and a default node 225. Next, she divides the web traffic among two organizational departments by creating a Department A node 226, and a Department B node 228. Each may have its own associated policy. Similarly, she divides TCP network bandwidth into separate traffic classes by creating a Department A node 230 and a Department B node 232. Each represents a separate traffic class which may have its own policy.

All traffic which does not match any user specified traffic class falls into an automatically created default traffic class which has a default policy. In a preferable embodiment, the default policy treats default class flows as unreserved traffic at the default (medium) priority. In Fig 2A, the default category is depicted by a default node 205, and in Fig. 2B, the default category is depicted by a default node 225.

2.2 <u>Creating Policies</u>

2.2.1 <u>Reserved vs. Unreserved</u>

Network managers create policies in order to assign service levels to traffic classes which are of particular interest. Service levels may be either reserved or unreserved. Reserved service is useful in instances where traffic 'bursts', i.e., sends chunks of data, or in interactive applications, which require a minimum bandwidth in order to function properly.

Reserved service is defined in terms of scaled rate, unreserved service is defined in terms of priority. Allocation of bandwidth resources by scaled rate is accomplished in a process known as "speed scaling" wherein the speed of the underlying network link upon which the flow is carried is detected and available bandwidth is allocated based upon this detected speed. Allocation of bandwidth resources for unreserved service is priority based. The total bandwidth resources for unreserved service demanded by all flows is satisfied according to a prioritized scheme, in which highest priorities are satisfied first, followed by successively lower priority applications. Unreserved service allocations are based upon the available bandwidth after reserved service demands have been satisfied.

Traffic may be a combination of reserved and unreserved services. For example, web traffic may have a service level defined at a web server for "*.html" to designate these types of files as traffic allocated unreserved service, as these are typically small files. In the same web server, "*.gif" files may be designated as a separate traffic class receiving reserved service. TCP connections and UDP sessions may be defined for reserved service.

2.2.2 Reserved Service: GIR vs. EIR

Reserved service may have both a guaranteed information rate (GIR) and an excess information rate (EIR) components. Guaranteed information rate represents a commitment of bandwidth resources to maintain a specified rate. Excess information rate is an allocation of bandwidth resources on an "as available" basis. Guaranteed information rate prevents evident lack of progress situations, such as stuck progress indicators, from occurring. Excess rate is allocated from available bandwidth, i.e. that which has not consumed by providing guaranteed rate service. As flows demand excess information rate at different priority levels, their demands are satisfied in order of priority level. For example, in an application having a ten voice over IP sessions, each session requiring a minimum of eight kilobits of bandwidth rate which must be guaranteed and up to 8 kilobits of additional bandwidth rate which may be used from time-to-time, would be allocated reserved service comprised of eight kilobits of guaranteed information rate for each of the ten sessions. Then, from the remaining available bandwidth resources, eight kilobits of excess information rate is allocated to each session based upon the priority of each session, until either all demands have been satisfied, or bandwidth resources are completely consumed.

When a guaranteed information rate cannot be provided to a requesting flow due to insufficient bandwidth resources, an admissions policy is invoked. The admissions policy may refuse a connection, display an error message or a combination of both. Admissions policies prevent service 'brown outs' which would otherwise occur if guaranteed information rate demand exceeds available bandwidth resources. By contrast, excess information rate requirements are not guaranteed. If a flow's request for excess information rate is not able to be satisfied from the available bandwidth resources, it is simply not met. In the previous example, if insufficient bandwidth existed in order to accommodate the ten voice over IP sessions, an admissions policy would be invoked.

Excess information rate (EIR) is redistributed each time a new flow is created, an existing flow is terminated, or a substantial change in demand for unreserved priority level occurs. By contrast, Guaranteed information rate (GIR) is allocated only at the time of creation of a flow and freed when the flow terminates.

In a preferable embodiment, reserved TCP service levels may be enforced by TCP Rate Control for both inbound and outbound traffic. TCP Rate Control is a technique well known to persons of ordinary skill in the art which scales reserved flows as excess rate is available without incurring retransmission.

2.3 Isolating Bandwidth Resources for Traffic Classes

Bandwidth resources may be dedicated to certain traffic classes. There are two degrees of isolation. *Hard isolation* creates an entirely separate logical channel for designated traffic classes. For example, department A and department B could both be hard isolated, setting up two logical .75 Mbps channels on a T1 access link, one for each department. With hard isolation, unused bandwidth in either channel would not be shared with the other department's traffic.

Hard isolation involves allocating arbitrary subsets of access link bandwidth for any traffic class or family of traffic classes. It completely protects the traffic belonging to the respective class or classes, but this protection comes at the cost of forsaken opportunities to share unused bandwidth.

Soft isolation allows sharing of unused bandwidth, but limits the amount of guaranteed bandwidth that can be allocated to any particular traffic class at any one time. This provides a basic level of fairness between traffic classes which have corresponding guaranteed service policies. But this fairness does not come at the cost of efficiency gains accrued from sharing available bandwidth.

3.0 System Function

3.1 Building a Traffic Classification Tree

Fig. 2C depicts a flowchart 203 showing the process steps in building a classification tree, such as classification tree 201 of Fig. 2A, by adding traffic class nodes, such as traffic class node 206, in order of specificity of classes. In a step 240, a new tclass node is allocated for the traffic class which is to be added to the classification tree. Next, in a step 242, a traffic specification is inserted for the new traffic class node, allocated in step 240. Then, in a step 244, the new tclass node is inserted into the classification tree. Next, in a step 246, bandwidth resources allocated based upon the traffic class hierarchy is redistributed to reflect the new hierarchy.

Fig. 2D depicts a flowchart 205 showing the component steps of traffic specification creation step 242 of Fig. 2C. The processing steps of flowchart 205 insert a

traffic specification, or tspec, into the traffic class allocated in step 240. In a step 250, a current tspec is initialized to the first tspec in the tclass. Next, in a decisional step 252, a determination is made whether the new tspec is less specific than the current tspec, initialized in step 250. If this is not the case, then in a step 258, the new tspec is inserted after the current tspec, and processing returns. Otherwise, in a decisional step 254, a determination is made whether this is the final tspec in the traffic class. If this is so, then in step 258, the new tspec is inserted after the last (current) tspec. Otherwise, in a step 256, the next tspec in the traffic class is examined by repeating processing steps 252, 254, 256 and 258.

Fig. 2E depicts a flowchart 207 showing the component steps of traffic specification creation step 244 of Fig. 2D. The processing steps of flowchart 207 insert the traffic class, or tclass, allocated in step 240 into the classification tree. In a step 260, a current sibling is initialized to the first sibling in the parent tclass. Next, in a decisional step 262, a determination is made whether the new tclass is less specific than the current sibling, initialized in step 260. If this is not the case, then in a step 268, the new tclass is inserted after the current sibling, and processing returns. Otherwise, in a decisional step 264, a determination is made whether this is the final sibling in the parent tclass. If this is so, then in step 268, the new tclass is inserted after the last (current) tclass. Otherwise, in a step 266, the next sibling in the parent traffic class is examined by repeating processing steps 262, 264, 266 and 268.

3.2 Allocating Bandwidth based upon Policy

Fig. 3 depicts a functional block diagram 301 of component processes in accordance with a particular embodiment of the present invention. Processing of a new flow, such as new flow 300 as described by a flowchart 501 in Fig. 5A, begins with a TCP autobaud component 302, which implements the automatic detection of flow speeds as depicted in Fig. 1E. TCP autobaud component 302 determines values for selectable information such as flow data rate of new flow 300. A classifier 304 determines the class for the new flow using the information determined by TCP autobaud 302. Having classified the traffic, the classifier next determines an appropriate policy for this particular flow from a classification tree (not shown). Based upon the policy, the classifier then determines a service level for the new flow. A bandwidth manager 306 uses the policy determined by classifier 304 in order to allocate bandwidth according to the service level prescribed by the policy.

Fig. 4A depicts data structures in a particular embodiment according to the invention. The bandwidth management aspect of the present invention is performed between inside clients and servers and the access link. Outside clients and servers are on the other side of the access link. In the embodiment of Fig. 4A, Traffic is directional. Inbound (flowing from an outside host to an inside host) traffic is managed entirely separately from outbound traffic.

A connection hash table 402 facilitates indexing into a plurality of TCP connections, each TCP connection having an associated flow, such as flows 300, 299 and 298 of Fig. 3. A plurality of TCP control blocks 404 are used to track connection-related information for each TCP connection. Host-related information is tracked using a plurality of host information control blocks 406, indexed by a host hash table 408. Information about particular hosts corresponding to TCP connections is tracked using the TCP control block 404.

Each TCP connection represented by TCP control block 404 has associated with it a gear 410 representing an outbound flow and a gear 411 representing an inbound flow. Each gear is an internal data object associated with a particular TCP connection or UDP session. Gears are protocol independent and contain rate-based information. Since network resources are managed separately for inbound and outbound traffic in this particular embodiment, there is a bandwidth pool 412 for outbound traffic and a bandwidth pool 413 for inbound traffic. Bandwidth pools represent collections of available network resources managed by bandwidth manager 306. Bandwidth pools are arbitrarily divided into a partition 414 in the outbound path, and a partition 415 in the inbound path. Each partition represents an arbitrarily-sized portion of available network resource. Gears become associated with particular partitions during the beginning of flow management for the new flow in step 518 of Fig. 5A. A policy 416 in the outbound path and a policy 417 in the inbound path become associated with partition 414 in the outbound path and partition 415 in the inbound path during the beginning of management of a new flow, as shown in Fig. 5A. These policies are associated with particular flows according to the traffic class that classifier 304 assigns to the particular flow.

Fig. 5A depicts flowchart 501 of the steps taken in initiating a new flow in accordance with a particular embodiment of the invention. In a step 502, TCP autobaud component 302 detects a TCP data rate of new flow 300. In a step 504, classifier 304 classifies the new flow to determine a policy. Classification is described in greater detail in Fig. 5F herein below. In a step 506, based upon the policy determined in step 504, a guaranteed information rate (GIR) and an excess information rate (EIR) are determined and speed scaled to the incoming TCP data rate, detected in step 502. Speed scaling is described in greater detail in Figs. 5G and 5H herein below. In a step 508, bandwidth manager 306 allocates guaranteed information rate resources and excess information rate resources to accommodate an initial target rate for the new flow. In a decisional step 510, bandwidth manager 306 determines if a partition limit would be exceeded by the allocation determined in step 508. If this is the case, then in a step 512, an admissions policy is invoked. Otherwise, in a step 514, the excess information rate resources demanded by new flow 300 is added to a total excess information rate resources demanded for this bandwidth pool. In a step 516, excess information rate resources are redistributed with respect to all flows within the partition, if demand has changed substantially since the last time it was redistributed. Fig. 5E depicts a flowchart 509 showing the steps in reapplying demand satisfaction parameters to all traffic flows in the bandwidth pool. A new target rate and total excess information rate resources are calculated. Finally, in a step 518, the new flow is associated with a partition.

Fig. 5B depicts a flowchart 503 showing processing of a packet belonging to an existing flow such as a forward flow packet 299. In a decisional step 520, a scheduler 308, using policy information determined at flow initiation, filters TCP acknowledgments for scheduling in a step 522. Output times are scheduled on millisecond boundaries, based upon the target data rate computed in step 508 of Fig. 5A. In as step 524, the remaining packets are delivered immediately to TCP, which performs just in time delivery.

Fig. 5C depicts a flowchart 505 showing processing by bandwidth manager 306 of a packet at the termination of a flow, such as an end flow packet 298. In a step 530, guaranteed information rate resources are restored. In step 532, excess information rate resources demanded by the terminating flow 298 is subtracted from the total excess information rate resources demanded. In a step 534, excess information rate resources are redistributed as described in Fig. 5D. Finally, in a step 536, the flow is removed from its partition.

Fig. 4B depicts specific data structures used in redistributing unreserved service resources among multiple flows in a particular partition. Input rates of particular flows, such as new flow 300, are measured by TCP autobaud component 302 and used to determine an Extrapolated Unreserved Demand or (EUD) 424. Flows requiring excess information rate resources, such as flows 299 and 298, having gears 410 and 411 (not shown), respectively, have associated with them an individual flow demands 420 and 421. The individual flow demands of all flows in a particular partition are added together to form an Aggregate Rate Demand (ARD) 422 at each priority level. Aggregate rate demand is added to extrapolated unreserved demand 424 to arrive at a total instantaneous demand 426. The total instantaneous demand is satisfied from the highest priority, priority 7, downward based upon the available excess information rate resources in the partition 414. This available rate is information rate which has not been used to satisfy guaranteed information rate demands. From the available excess information rate resources, a satisfaction percentage vector 428 is computed having the percentage of each priority of flow which may be satisfied in the total demand vector 426 by the available excess information rate resources in partition 414. Individual flow demand vectors 420 and 421 are each multiplied by the satisfaction percentage for each priority 428 in order to arrive at a target rate which is incorporated in gears 410 and 411 corresponding to each flow. The satisfaction percentage vector 428 is also applied to the Extrapolated Unreserved Demand (EUD) 424 to determine unreserved targets.

3.3 <u>Reconciling Needs for Bandwidth</u>

Fig. 5D depicts flowchart 507 of the processing steps to redistribute excess information rate resources with respect to reserved service flows within a given partition.

In a step 540, a demand satisfaction metric is computed for the bandwidth pool being redistributed. In a decisional step 542, a determination whether the demand satisfaction metric computed in step 540 has changed substantially since the last time demand was redistributed. If this is the case, then in a decisional step 544, a loop is entered for each gear associated with the partition owning the bandwidth pool being redistributed. In a step 546, demand satisfaction parameters are reapplied a traffic flow associated with each gear and a new target rate and a new total excess information rate resources are calculated. Processing continues until in decisional step 544, it is determined that all gears have been processed.

Fig. 5E depicts flowchart 509 of the processing steps for reconciling bandwidth among a plurality of flows, requiring reserved service and unreserved service resources. For example, partition 414 of Fig. 4B has unreserved service flow 300 and reserved service flows 299 and 298. In a decisional step 550, a loop is formed with a step 554, in which the individual flow demands of reserved service flows, such as flows 299 and 298 in partition 414, are added together to form an aggregate rate demand (ARD) 422. Next, in a step 552, an extrapolated unreserved demand (EUD) 424 is determined from the input rate of unreserved service traffic 300 in partition 414. In a preferable embodiment, the EUD must be inflated in order to promote growth in traffic. This is done to accommodate the elastic nature of TCP traffic. Next, in a step 556, the EUD determined in step 552 and the ARD determined in step 554 are combined to form an instantaneous total demand (TD) 426 for the entire partition 414. The total instantaneous demand is satisfied from the highest priority level, level 7, downward based upon the available unreserved resources in the partition 414. In a step 558, a satisfaction percentage vector (SP) 428 is computed for the partition from the total demand vector (TD) 426 determined in step 556. The satisfaction percentage indicates that percentage of the demand at each priority level that can be satisfied from the unreserved resources in the partition. Next, in a step 560, a target rate for reserved rate based flows, such as flows 299 and 298, is computed. Finally, in a step 562, a target rate for unreserved priority based flows, such as flow 300, is computed.

3.4 Searching a Classification Tree

Fig. 5F depicts a flowchart 511 showing the component steps of traffic classification step 504 of Fig. 5A. The processing steps of flowchart 511 determine a class and a policy for a flow, such as new flow 300, by traversing a classification tree such as the classification tree 201 in Fig. 2A. Processing begins with a first node in the classification tree 201, such as department A node 202. In a step 570, a traffic specification of a child node, such as FTP node 206 is compared with a flow specification of the new flow 300. In a decisional step 572, if a match is discovered, then in a step 574, the processing of flowchart 511 is applied to the child node 206 recursively. Otherwise, if child node 206 does not match the new flow, then in a decisional step 576, processing determines whether any sibling nodes exist for the child node 206. If processing detects the existence of a sibling of child node 206, such as child node 208, then processing continues on node 208 with step 570. Otherwise, if there are no further child nodes for node 202, then in a decisional step 578, a determination is made whether the node is a leaf node having a policy. If no policy exists, then in a step 580, processing backtracks to a parent node and looks for a policy associated with the parent node to apply to the new flow 300. Otherwise, if a policy is associated with the node 202, then in a step 582, the policy is associated with the new flow 300.

3.5 Speed Scaling

Fig. 5G depicts a flowchart 513 showing the component steps of speed scaling step 506 of Fig. 5A. The processing steps of flowchart 513 determine an acceptable allocation of bandwidth to reserved service flows, i.e., GIR and EIR, by allocating rate for each gear, such as gears 410 and 411, based upon individual flow demands, base limits and total limits. In a step 582, scaled rate for GIR is set based upon a connection speed detected by TCP autobaud component 302 for a particular flow, such as flow 299, having a gear, such as gear 410. Then in a step 584, scaled rate for EIR is set based upon a connection speed detected by TCP autobaud component 302 for a particular flow, such as flow 298, having a gear, such as gear 411. Next, in a step 586, a limit is computed from either the available EIR remaining after the allocation to flow 299 in step 584, or from a total limit, if such a limit is specified. Then, in a step 588, a determination is made whether any extra EIR resource remains after the limit computed in step 586 has been satisfied. Then, in a step 590, limits computed in step 586 are allocated to flows, such as flow 299 and flow 298, in order of descending priority level.

In a decisional step 592, the processing of steps 582 through 592 are applied to subsequent gears, if any.

Fig. 5H depicts a flowchart 515 showing the component steps of scaled rate setting steps 582 and 584 of Fig. 5G. The processing steps of flowchart 515 determine whether the rate of the underlying link detected by TCP autobaud 302 exceeds limits imposed on the network. In a decisional step 592, a determination is made whether the detected speed of a flow is less than a base limit minimum. If this is so, then in a step 594, the base limit is returned as the scaled rate. Otherwise, in a decisional step 596, a determination is made whether the total rate of all flows is greater than a maximum limit. If this is so, then in a step 598, the maximum limit is returned as the scaled rate. Otherwise, in a decisional step 600, a determination is made whether the detected speed of a flow is greater than a total rate of all flows. If this is so, then in a step 602, the maximum limit is returned as the scaled rate. Otherwise, in a step 604, a percentage of the total rate, represented by the maximum limit, is computed. Then, in a step 606, the percentage computed in step 604 is applied to a total rate available, i.e., the maximum limit - base (minimum) limit. The result is returned as the scaled rate.

4.0 Conclusion

In conclusion, the present invention provides for a policy based bandwidth allocation method for IP packet telecommunications systems wherein bandwidth is allocated to requesting flows according to automatically determined application requirements. An advantage of network management techniques according to the present invention is that network managers need only define traffic classes which are of interest. A further advantage of the present invention is that traffic classes may include information such as a URI for web traffic. A yet further advantage of the present invention is that service levels may be defined in terms of explicit rates, may be scaled to a remote client or server's network access rate, specified for high speed and low speed users and defined in terms of a guaranteed minimum service level.

Other embodiments of the present invention and its individual components will become readily apparent to those skilled in the art from the foregoing detailed description. As will be realized, the invention is capable of other and different embodiments, and its several details are capable of modifications in various obvious respects, all without departing from the spirit and the scope of the present invention.

Accordingly, the drawings and detailed description are to be regarded as illustrative in nature and not as restrictive. It is therefore not intended that the invention be limited except as indicated by the appended claims.

A pseudo code description of a particular embodiment is included herein below:

```
/*************************************************************************
**
 *
 *
 * description: BANDWIDTH POOL related functions.
 *

*************************************************************************
**
 */

#define EIR_LEVELS                     8
#define EIR_HIGHEST                    7
#define EIR_LOWEST                     0
#define EIR_DEFAULT                    3
#define EIR_NONE                      -1
#define EIR_ALL_LEVELS          (0xffff)

/*
 * An EIR vector is a demand for Excess rate bandwidth at each of
 * the N eir prioritylevels.
 */
typedef BPS                                     EIR_VECTOR[EIR_LEVELS];

/*
 * Rate Specifier, i.e. CIR + EIR...
 */
typedef struct {
 BPS               cirLimit;              /* How much CIR I want */
 BPS               eirDemand;               /* total of eirDemandvector */

 EIR_VECTOR        eirDemandVector;         /* How much EIR I want */
 /*
  * UDP stuff tbd...
  */
 INT32                     delayBound;
} RATE_SPEC;

#define RATE_UNCONSTRAINED 0xffffffffL

/*
 * Latency status record
 */

#define LATENCY_BIG_TROUGH 0xffffffffUL

typedef struct {
 INT32             ewmaValuc;
 INT32             lastPeak;
 INT32              lastTrough;
 INT32              peak;
 INT32              trough;
 TIME        timeStamp;
} LATENCY;


typedef struct _gearshift {
 DLINK                      dlink;

#define GF_INITIALIZED                    0x0001
#define GF_SCALED                                0x0002
#define GF_ACTIVE                                0x0004
#define GF_EIR_OVERALLOCATED       0x0008        /* allocated though none
available*/
#define GF_CIR_OVERALLOCATED       0x0010        /* allocated though none
available*/
#define GF_QUIESCENT                             0x0020
#define GF_PRIORITY                              0x0040
#define GF_RATE                                  0x0080
#define GF_DQ                                           0x0100
#define GF_TCPRC                                 0x0200
#define GF_FILTER                                0x0400
#define GF_NEEDS_REALLOC                  0x0800
#define GF_CIR_COMMITTED                  0x1000

 void*              hosts[2];                                      /* Who the
hosts are! */
 INT16                      flags;
 INT16                      priCounted;
 BPS                target;
 INT32                      cirInUse;
 INT32                      eirInUse;
 BPS                lastScaledBps;
 RATE_SPEC rspec;
} GEARSHIFT;

/*
 * For scaling CIR to connection speed. See traffic/scale.c.
 * If limit is set to 0, no CIR is allocated.
 */
typedef struct {
 BPS base;
 BPS limit;
} CIR_SPEC;

/
 * For scaling EIR to connection speed, i.e. creating eirLimits vector.
 * hp -highest priority, hp-base, and hp-limit and optional total limit.
 * See traffic/scale.c.
 *
 * As with CIR Spec, if hpLimit is set to 0, no EIR is allocated.
 */
typedef struct {
 INT8 hp;
 BPS  hpBase;
 BPS  hpLimit;
 BPS  totalLimit;
} EIR_SPEC;




/***************************************************************************
**
 *
 * bwpRedistributeEir(bwp, force)
 *
 * Called when a FLOW is added to or deleted from a tclass belong to
 * this partition (partiton is a bandwidth pool - BW_POOL)
 *
 * Reallocate EIR. If new allocation percentages are significantly different
 * for any priority level, then spin through the GEARs to update the
 * SATISFACTION percentage level at each GEAR (at each TCLASS) with a
 * demand at that level.
 *
 * NOTE: Execution efficieny is critical.

****************************************************************************
**
 */
void
bwpRedistributeEir(BW_POOL *bwp, int forceAll)
{
 whereStr("bwpRedistributeEir");
 EIR_MASK changeMask = 0;
 EIR_VECTOR eirNewPercentage, urDemand, urAllocated;
 int i, available, urTotal, totalDemand, demand, diff;
 int new, old;
 int dir = bwp->flags & BWF_DIR_MASK;

 VALIDATE_BWP(bwp);

 available   = bwp->limit - bwp->cirCurrent;
 totalDemand = eirVectorSum(bwp->eirDemand);
 urTotal     = bwpUqDemand(bwp, urDemand);
```

```
bwp->stats.eirRedistributions++;
/*
 * All SATISIFIED ?
 */
if (available > (totalDemand + urTotal)) {
        eirVectorSet(eirNewPercentage, 100);
        bwp->eirCurrent = totalDemand;
        /*
         * And give leftovers to UQ with a little margin unallocated...
         */
        urTotal = max((available - totalDemand) - (bwp->limit >> 3), urTotal);
        bwpUqSetTarget(bwp, urTotal);
}
/*
 * None SATISIFIED ?
 */
else if (available <= 0) {
        /*
         * No EIR is allocated...
         */
        bwp->eirCurrent = 0;
        eirVectorSet(eirNewPercentage, 0);
        /*
         * Priority buckets set to minimal output rate...
         */
        bwpUqSetTarget(&bwp->uq, 0);
}
/*
 * Some SATISFIED...
 */
else {
        bwpSetEirPct(bwp, available, urDemand, eirNewPercentage);
        forceAll = TRUE;
}
/*
 * OK - now check NEW versus OLD percentages to see if we have to
 * run through the gears...
 */
if (forceAll)
        changeMask = EIR_ALL_LEVELS;
else {
        for (changeMask = i = 0; i <= EIR_HIGHEST; i++) {
                new = eirNewPercentage[i];
                old = bwp->eirPctSatisfied[i];

                if (new == old)
                        continue;
                else {
                        /*
```

```
                         * Check to see if the % difference is significant...
                         */
                        diff = abs(new - old);
                        if (diff > (old >> 3))          changeMask |= (1 << i);
                }
        }
 }
 eirVectorCopy(eirNewPercentage, bwp->eirPctSatisfied);

 if (changeMask) bwpUpdateGearEir(bwp, changeMask);
}


/************************************************************************************
**
 *
 * bwpSetEirPct(bwp, avail, urTotal, urDemand, newPctV)
 *
 *      Allocate available bandiwdth between UQ and EIR. Note only one
 * priority level is PARTIALLY SATISIFIED.

************************************************************************************
**
 */
void
bwpSetEirPct(BW_POOL *bwp,
                        int avail,
                        EIR_VECTOR urDemand,   /* Unreserved demand vector to be
filled in */
                        EIR_VECTOR newPct)              /* New satisfaction percentage
to be filled in*/
{
 whereStr("bwpSetEirPctOUT");
 int demand, i, pct, urPct, eirPct;
 EIR_VECTOR urAllocate;

 /*
  * Go through each prioirty level from hi to low...
  */
 for (i = EIR_HIGHEST; i >= 0; i--) {

        demand = bwp->eirDemand[i] + urDemand[i];
        /*
         * None left ?
         */
        if (avail <= 0){
                newPct[i] = 0;
                urAllocate[i] = 0;
        }
        /*
         * More than enough - 100% for both...
         */
        else if (avail >= demand) {
                newPct[i] = 100;
                urAllocate[i] = urDemand[i];
                avail -= demand;
        }
        else {
                /*
                 * Have to divide available here between UQ and reserved flow EIR...
                 */
                if (bwp->eirDemand[i] == 0){
                        urAllocate[i] = avail;
                        newPct[i] = 0;
                }
                else if (urDemand[i] == 0){
                        newPct[i] = (avail * 100)/demand;
                        urAllocate[i] = 0;
                }
                else {
                        /*
                         * Need to share. The overall level is satisfied N
                         * percent. UQ and rate EIR each get their proportional
                         * share of that N percent...
                         */
                        pct    = max(1, (avail*100)/demand);
                        urPct  = max(2, (urDemand[i]*100)/demand);
                        urPct  = min(98, urPct);
                        eirPct = 100 - urPct;
                        assert(eirPct >= 2 && eirPct <= 98);

                        newPct[i]       = max(2,(eirPct*pct)/100);
                        urPct           = max(2,(urPct*pct)/100);
                        urAllocate[i] = (urDemand[i] * urPct)/100;
                }
                /*
                 * Only one priority level requires this calculation -
                 * all lower levels will be 0 percent satisified...
                 */
                avail = 0;
        }
}

/*
 * Set the output target for the unreserved priority traffic based
 * on the total unreserved vector...
 */
bwpUqSetTarget(bwp, eirVectorSum(urAllocate));
```

```
}


/************************************************************************
**
 *
 * bwpUpdateGearEir(bwp, changeMask)
 *
 * Cycle through all GEARs using EIR at the specified priority levels and
 * change their SATISFACTION percentage allocations.
 *
 * NOTE: Execution efficieny is critical.

************************************************************************
**
 */
void
bwpUpdateGearEir(BW_POOL *bwp, EIR_MASK changeMask)
{
 whereStr("bwpUpdateGearEIR");
 TCLASS *tclass;
 GEARSHIFT *g;
 int i, tclassTotal, poolTotal;
 INT32 *evPct = bwp->eirPctSatisfied;
 INT32 *evActual, *evDemand;

 blurt2("pool %s mask %x", bwp->root->name, changeMask);
 bwp->stats.gearUpdates++;

 poolTotal = 0;
 /*
  * For all TCLASSs using this BW POOL...
  */
 for (tclass = bwp->tclassList; tclass != NULL; tclass = tclass->bwpThread) {
        /*
         * For all GEARs at this TCLASS...
         */
        tclassTotal = 0;

        for (g = tclass->activeList; g != NULL; g = (GEARSHIFT *)g->dlink.next) {
                /*
                 * Are any of the GEARs EIR levels affected - That is, does
                 * the flow have a demand at a changed Satisfaction Percentage
                 * priority level ?
                 */
                if (! IS_FLAG_SET(g->flags, GF_DEMANDS_EIR))
                        continue;

                if ((changeMask & g->rspec.eirActiveMask) == 0){
```

```
                tclassTotal += g->rspec.eirTotal;
                continue;
        }
        /*
         * Can we now allocate for an overalloced flow ?
         */
        if (IS_FLAG_SET(g->flags, GF_EIR_OVERALLOCATED)){
                blurt1("recovering from eir-overalloc tclass %s", tclass->name);
                tclassFreeOverallocEir(tclass, g->rspec.eirTotal);
                g->rspec.eirTotal = 0;
                g->flags &= ~GF_EIR_OVERALLOCATED;
        }
        /*
         * OK - allocate our N percentage levels. Note that no more
         * than 1 level is not zero or 100.
         */
        evActual = g->rspec.eirActual;
        evDemand = g->rspec.eirDemand;

        for (i = 0; i <= EIR_HIGHEST; i++) {
                if ((changeMask & (1 << i)) == 0) continue;

                if (evPct[i] == 100)
                        evActual[i] = evDemand[i];
                else if (evPct[i] == 0)
                        evActual[i] = 0;
                else
                        evActual[i] = (evDemand[i] * evPct[i])/100;

        }
        /*
         * Note we need to update g->rspec.eirTotal and g->target
         */
        g->rspec.eirTotal = eirVectorSum(g->rspec.eirActual);
        tclassTotal += g->rspec.eirTotal;
        /*
         * And set target rate...
         */
        g->target.bps = g->rspec.eirTotal + g->rspec.cirTotal;
        /*
         * Check too see if we got ALL of our bandwidth taken away.
         * If so OVERALLOCATE some...
         */
        if (g->target.bps == 0) bwpOverAllocEir(bwp, tclass, g);
        /*
         * Update TCLASS total...
         */
        tclass->eirCurrent = tclassTotal;
        } /* For all GEARs */

        poolTotal += tclassTotal;

 } /* For all TCLASSs */

 bwp->eirCurrent = poolTotal;  /* Set the total EIR in use for the pool */

}


/*****************************************************************************
**
 *
 * bwpUqDemand(bwp)
 *
 * Return current 'demand' for unreserved bandwidth, EXTRAPOLATED from
 * current output and demand rates...

*****************************************************************************
**
 */
int
bwpUqDemand(BW_POOL *bwp, EIR_VECTOR demandV)
{
 whereStr("bwpUqDemand");
 UQ *uq = &bwp->uq;
 BPS demandRate = 0;
 int old = uq->target;
 int i = 0;
 BPS outputRate = exponential weighted moving average of uq->outputRate

 if ((bwp->flags & BWF_DIR_MASK) == DIR_INBOUND){
        eirVectorSet(demandV, 0);
        return(outputRate);
 }

 for (i = EIR_HIGHEST; i >= 0; i--){
        demandV[i] = exponential weighted moving average of uq->demandRate[i]

        Add overstatement factor to demand at this level

        demandRate += demandV[i];
 }
 demandRate = max(outputRate, demandRate);
 uq->target = min(demandRate, bwp->limit);
 blurt2("new %ld old %ld", uq->target, old);
 return(uq->target);
}
```

TCB - transport control block - TCP State information for both directions

```
/*
 * BCB - Buffer Control Block. Contains packet info, including parsed
 * flow spec, as well as pointers to various layers of the actual
 * packet buffer.
 */
#define BCB_BUF_SIZE      1548
#define BCB_MAGIC                   0xdeadbeef

typedef struct _bcb {
 struct _bcb     *next;
 struct _bcb     *last;

 INT32                          linkHeaderLen;                     /* length of link header
                                                      */
 int                   frameMaxSize;                      /* Actual buffer memory
          */
 INT8                  *frameStart;                /* Note a pointer to the TOP of the */
                                                                                    /* bcb is
embedded at frameStart-4  */
 INT8                  *dataStart;                        /* Current Data pointer and
length */
 int                   dataLen;
 USEC_TIME  tick;                                         /* time the packet should be
output and
                                                                                    /*
timestamp of when it was sent                             */
 FLOW_SPEC             fspec;                             /* Parsed FLOW description...

 INT8                  *protHdrStart;                     /* Pointer to and length of
whatever is      */
 INT16                          protLength;                        /* Encapsulated in the
IP packet.               */
 OBJECT_ID  tclassId;                              /* class ID for accounting purposes
        */

}BCB;

/*
 * FLOW components and object.
 */
typedef unsigned char DIRECTION;

#define DIR_INBOUND              0
#define DIR_OUTBOUND             1
```

```
typedef struct {
 INT16 port;
 INT32 host;
} SOCKET;

typedef struct {
 SOCKET inside;
 SOCKET outside;
} CONN;

typedef struct _flow_spec {
 CONN               conn;

#define PF_INET          0
#define PF_IPX                   1
#define PF_ATALK         2
#define PF_NETBIOS       3
#define PF_SNA                   4
#define PF_FNA                   5

 INT8          protFamily;
 INT8          tos;
 INT8          ipProtocol;
 INT8          direction;

 ServiceIndex  service[NUM_DIRECTIONS];
 INT16                uriLen;
 char          *uri;
} FLOW_SPEC;

/*
 * For storing URIs
 */
#define MAX_URI_NAME_LEN     128

typedef struct _uri_buf {
 struct _uri_buf        *next;
 char                              name[MAX_URI_NAME_LEN];
} URI_BUF;

/*
 * For matching URIs
 */
typedef struct {

#define URI_FLAG_QUERY                  (0x01)              /* '?' present?
*/
#define URI_FLAG_SPECIAL_EXACT     (0x02)         /* special-case exact-match? */
#define URI_FLAG_SPECIAL_LEFT      (0x04)       /* do special-case left-match?
*/
#define URI_FLAG_SPECIAL_RIGHT     (0x08)       /* do special-case right-match?
*/

 unsigned char flag;                                     /* URI_FLAG_*
bitmask */
 unsigned char count;                           /* no. of segments in uriBuf[]
*/

 URI_BUF              *uriBufCompile;                    /* the compiled URI (if
non-nil) */
 URI_BUF              *uriBufLiteral;                    /* the original URI (if
non-nil) */

} URI_SPEC;


/*
 * Define Traffic Class Specifier...
 */
typedef struct _t_spec {
 FLOW_SPEC fspec;
 FLOW_SPEC fspecMask;
 URI_SPEC  uriSpec;                   /* present if uriSpec.uriBufCompile != 0 */
} TSPEC;


/*
 * TCLASSs are directional. There is both an INBOUND and OUTBOUND
 *       root partition. NOTE this accomodates ASYMMETRIC links.
 *       INBOUND partition refers to direction of data flow, not of connection
 *       origin
 *
 * TCLASSs have a component specifications (lists of TSPECS) which is
 * ordered by increasing specificity. The global flow implicitly matches
 * all traffic classes.
 *
 * TCLASSs are organized into an N-ARY tree, where children of tclasses
 * are arranged in order of exception, normal, and inheritable.
 *
 *       Sorting/searching order is
 *               exception tclasses without applied policy
 *               exception tclasses with applied policy
 *               tclasses without policy
 *               tclass   with policy
 *               inheritable tclass with policy (inheritable has policy by definition)
 *
 *       within each group, tclasses are ordered by Decreasing Match Specificity
```

```
*
* A FLOW is matched to a policy as follows:
*               Starting at the global partition (implict component match),
*
*                       a MATCHALL tspec/policy exists at the top level for both the
*                       inbound and outbound tree.
*
*                       Intermediate traffic classes (those with children) do NOT
*                       have associated policies.
*
*                       tclass_check(tclass, flow)
*
*                               tclass are checked in above specified order against flow.
*                               If they match, recurse to call tclass_check(match_tclass,
flow)
*
*                               OK - we have found the most specific tclass. If it has an
*                               associated policy, we are done. Else we need to check back
up
*                               thorugh the tree for an inherited policies.
*
*                               Checking for inherited policies goes up to parent, and then
*                               down through parents remaining siblings, then recurses up
to
*                               parents parent, etc... till a match is found.
*/

typedef struct _tclass {
 struct _tclass         *sibling;
 struct _tclass         *child;
 struct _tclass         *parent;

#define TCF_INBOUND                             DIR_INBOUND
#define TCF_OUTBOUND                            DIR_OUTBOUND
#define TCF_DIR_MASK                            0x0001
#define TCF_ACTIVE                              0x0002
#define TCF_IDLE                                0x0000
#define TCF_GLOBAL                              0x0004
#define TCF_DISABLED                            0x0008

#define TCF_EXCEPTION                   0x0010
#define TCF_INHERITABLE                         0x0020
#define TCF_STANDARD                            0x0000
#define TCF_TYPE_MASK                   (TCF_EXCEPTION+TCF_INHERITABLE)

#define TCLASS_TYPE(t)                  ((t)->flags & TCF_TYPE_MASK)

#define TCF_WEB                                 0x0100
#define TCF_AUTO_CLASSIFYING            0x0200
#define TCF_AUTO_CREATED            0x0400

 INT16                              flags;
 INT8                         direction;
 OBJECT_ID              id;
 INT32                              hits;

 /*
  * Rate pool stuff ... Need to have EIR limits/priorities as well
  * as CIR limits..
  */
 PKT_ACCUM                    pktAccum;
 EWMA_STATS                   traffic;


 TSPEC_ITEM                   *imsTspecList;                    /* IMS
ordered TSPEC ITEMs        */
 POLICY_ITEM                  *policy;                              /*
May be NULL */
 /*
  * Points to self or root class is CIR set or inherited or NULL if
  * CIR is not enforced...
  */
 struct _tclass  *cirRoot;
 BPS                          cirRootLimit;
 BPS                          cirRootCurrent;
 /*
  * Values for this Tclass...
  */
 BPS                          cirCurrent;
 BPS                          eirCurrent;
 BPS                          cirOverAllocated;
 BPS                          eirOverAllocated;
 BPS                          cirCommitted;                 /* Amount of
CIR committed for outstanding HTTP request */
 LATENCY                      xactLatency;

 CIR_STATS              cirStats;
 /*
  * List of active sessions/connections...
  */
 DLINK                              activeList;

 struct _bw_pool       *bwp;
 struct _tclass  *bwpThread;

 char                         name[SYS_OBJ_NAME_LEN];

} TCLASS;
```

```
/*
 * Abstracted packet path...
 */
main_pkt_path()
{

 if its an IP input packet, it is parsed with parseTspec to
 embed an fspec inside the bcb

 the packet is processed by the matching protocol
 finite state machine
}

/*************************************************************************
**
 *
 * ipParseFlow(bcb)
 *
 *       Build basic flow spec for the given input packet. Just parse basic ip
 * info and leave headers in net format.

*************************************************************************
**
 */
int
ipParseFlow(BCB *bcb)
{
 FLOW_SPEC *fspec = &bcb->fspec;
 CONN *conn = &fspec->conn;
 INT8 *cp = NULL;
 INT16 len, hdrLen;

 memset(fspec, 0, sizeof(FLOW_SPEC));
 fspec->protFamily = PF_INET;

 /*
  * Extract IP source and destination and protocol type from net header.
  */
 if (bcb->device == INSIDE_IFACE) {
         conn->inside.host = get32(&bcb->dataStart[12]);
         conn->outside.host = get32(&bcb->dataStart[16]);
 }
 else {
         conn->inside.host = get32(&bcb->dataStart[16]);
         conn->outside.host = get32(&bcb->dataStart[12]);
 }
 fspec->ipProtocol = bcb->dataStart[9];
 fspec->tos = bcb->dataStart[1];
 /*
  * Advance to next protocol header...
  */
 hdrLen = ((bcb->dataStart[0] & 0x0f) << 2);
 cp = bcb->dataStart + hdrLen;
 /*
  * If it is TCP or UDP extract ports...
  */
 if (fspec->ipProtocol == TCP_PTCL || fspec->ipProtocol == UDP_PTCL) {
         if (bcb->device == INSIDE_IFACE) {
                 conn->inside.port  = get16(cp);
                 conn->outside.port = get16(cp + 2);
         }
         else {
                 conn->inside.port  = get16(cp + 2);
                 conn->outside.port = get16(cp);
         }
 }
 bcb->protHdrStart = cp;

 len = get16(&bcb->dataStart[2]);
 bcb->protLength = len - hdrLen;

 fspec->direction = (bcb->device == INSIDE_IFACE) ? DIR_OUTBOUND :
DIR_INBOUND;

 return(OK);
}
```

NOTE: WEBNESS may be set for a connection by virtue of that connection being on the well known HTTP port (80), or by an explicit user configuration indicating that a certain port if for a web server, or by automatic recognition process which detects an http 'signature' in traffic destined towards a certain repetitively used (server) port.

```
tcp_finite_state_machine()
{
 handles connection establishment and clearing

 for data on an established connection, if
 connection has WEBNESS, check to see if it an
 http request.

}
```

```
/****************************************************************************
**
 *
 * httpParseRequest(tcb, bcb, tcpHdr)
 *
 *       If we find a new URI from a GET operation, re-lookup the policy for the
 * OTHER direction flow. So the HTTP BURST will effectively start from the
 * time of the GET. When the actual data burst starts in the other
 * direction, this policy will already be set, i.e. tmgrSetPolicy() will
 * do nothing.
 *

****************************************************************************
**
 */
int
httpParseRequest(TCB *tcb, BCB *bcb, TCP_HDR *tcpHdr)
{
 whereStr("httpParseRequest");
 int dir = bcb->fspec.direction;
 int len, rc;
 CONN *conn = &bcb->fspec.conn;
 char *uri, *method;
 HALF_CONN_INFO *hci = &tcb->halfConns[dir];
 HALF_CONN_INFO *otherHci = hci->other;

 /*
  * Is it a GET or POST ?
  */

 if ((len = httpGetUri((char *)tcpHdr->dataStart, tcpHdr->dataLen, TRUE, &uri,
&method)) > 0) {

        /*
         * Set URI pointer in flowspec embedded in BCB...
         */
        bcb->fspec.uri = uri;
        bcb->fspec.uriLen = len;

        /*
         * httpSetPolicies will CLASSIFY this URL and find policy...
         */
        rc = httpSetPolicies(tcb, bcb, tcpHdr);

        if (rc == NOTOK) {
                admitFailHttpGetRequest(tcb, bcb, tcpHdr);
                return NOTOK;
        }
        /*
```

```
         * Check for Admission Control...
         */
        if (admitHttpGet(tcb, dir, bcb, tcpHdr, (rc == OK_BUT_NO_CIR)) != OK)
                return NOTOK;

        bcb->fspec.uri = NULL;
        /*
         * This is the start of a burst on the other connection direction (HCI).
         */
        otherHci->burstState = HCF_BURST_WAITING_FOR_REPLY;
        otherHci->reqStamp = sysTimeStamp();
 }
 else {
        if (method && httpValidateMethod(method)){
                httpFsmMibOtherOps++;

                if (tmgrInitTcpHalf(tcb, dir, NULL, 0) != OK ||
                        tmgrInitTcpHalf(tcb, OTHER_DIRECTION(dir), NULL, 0) !=
OK){

                        admitFailHttpGetRequest(tcb, bcb, tcpHdr);
                        return(NOTOK);
                }
                }
 }

 return(OK);
}

/*************************************************************************
**
 *
 * httpGetUri(pkt, pktLen, getOrPostOnly, pUri, pMethod)
 *
 * Check for a valid HTTP request. If valid, parse URI and method.

*************************************************************************
**
 */
int
httpGetUri(const char *httppkt,
                int pktlen,
                int getOrPostOnly,
                char **pUri,
                char **pMethod)
{
 whereStr("httpGetUri");
  char *header, *protocol, *method, *cp;
  int headerlen, methodlen, plen, urilen;
```

```
*pMethod = NULL;
  method = NULL; header = NULL; protocol = NULL;
  headerlen = methodlen = plen = urilen = 0;

  headerlen = getline(httppkt,pktlen,&header);

  if ((header == NULL) || (headerlen <= 0))        return (-1);

  method = header;
  method += strspn(method," \t");

if (method == NULL)
        return (-1);

*pMethod = method;

if (getOrPostOnly){
        if (strncmp(method,"GET",3) != 0 && strncmp(method,"POST",4) != 0)
                return (-1);
}
else if (httpValidateMethod(method) == 0)
        return (-1);

  *pUri = strpbrk(method," \t");
  methodlen = *pUri - method;
  *pUri += strspn(*pUri," \t");

  protocol = strpbrk(*pUri," \t");
  urilen = protocol - *pUri;

  cp = strpbrk(protocol,"\r");
  plen = cp - protocol;
  cp += strspn(cp,"\r\n");

  if (plen <= 0)
  {
        attn("no protocol");
        urilen = 0;
}
  return urilen;
}

/***************************************************************************
**
 *
 * httpSetPolicies(tcb, getFspec)
 *
 *
```

```
*************************************************************************
**
 */
httpSetPolicies(TCB *tcb, BCB *bcb, TCP_HDR *tcpHdr)
{
 int                     dir = bcb->fspec.direction;
 HALF_CONN_INFO          *otherHci = &tcb->halfConns[OTHER_DIRECTION(dir)];
 TCLASS*                 tclass;
 POLICY_ITEM*      pi;
 HALF_CONN_INFO          *hci = otherHci->other;
 int                     setResult, tos;

 /*
  * Set any service abstraction in bcb.fspec...
  */
 serviceSetFlowSpec(&bcb->fspec, tcb->service, tcb->flags & TF_DIR_MASK);


 bcb->fspec.direction = OTHER_DIRECTION(dir);        /* need correct direction for
autoclassification */

 setResult = tmgrSetPolicy(&otherHci->policyId,
                                                     &otherHci->tclassId,
                                                     &bcb->fspec,
                                                     tcpSpeedGet(tcb),
                                                     &otherHci->gears,
                                                     tcb->hosts,
                                    TRUE);

 bcb->fspec.direction = dir;

 TCB_SET_POLICY(tcb, OTHER_DIRECTION(dir));
 /*
  * Now initialize the client->server direction.
  */
 if (tmgrInitTcpHalf(tcb, dir, bcb->fspec.uri, bcb->fspec.uriLen) == NOTOK)
        return NOTOK;

 return (setResult == OK) ? OK : OK_BUT_NO_CIR;
}


/*************************************************************************
**
 *
 * tmgrSetPolicy(pidPtr, tclassIdPtr, fspec, bps, gears, hosts)

    Called at the appearance of a BURST or an HTTP REQUEST. Called by
 * both TCP and UDP state machines.
 *
 * Set allowed CIR/EIR based on current link state, traffic class, and
 * connection speed...
 *

****************************************************************************
**
 */
int
tmgrSetPolicy(OBJECT_ID *pidPtr,
                        OBJECT_ID *tclassIdPtr,
                        FLOW_SPEC *fspec,
                        SPEED bps,
                        GEARSHIFT *g,
                        void *hosts,
                        int forceHits)
{
 whereStr("tmgrSetPolicy");
 POLICY_ITEM       *oldPi = NULL;
 POLICY_ITEM *pi = NULL;
 TSPEC_ITEM        *ti;
 OBJECT_ID  oldPid = *pidPtr;
 OBJECT_ID  oldTclassId = *tclassIdPtr;
 TCLASS            *tclass = NULL;
 BOOL                     newUri;
 BOOL                     newPolicyOrTclass = FALSE;
 int               latency, confidence, retval;

 /*
  * Do we already have a policy and tclass set ? could be a re-lookup, e.g.
  * persistent HTTP second click...
  */
 if (pidPtr->val != 0)
              pi = oldPi = policyIdToPolicy(*pidPtr);
 if (tclassIdPtr->val != 0)
        tclass = tclassIdToTclass(*tclassIdPtr);

 /*
  * If it is URI based, we may be changing policy/tclass. So we
  * need to free any bandwidth here we may currently own.
  */
 newUri = (fspec->uri != NULL);
 /*
  * Lookup Traffic Spec if policy is valid...
  */
 if (newUri || oldPi == NULL || tclass == NULL) {
        newPolicyOrTclass = TRUE;
        pMatchSpecial(fspec);
        /*
         * Find the class and policy for a flow in this direction...
         */
        pi = tclassCheck(fspec, GlobalTclasses[fspec->direction], &tclass, &ti);
        *pidPtr    = pi->id;
        *tclassIdPtr = tclass->id;
        doAccounting = TRUE;
}

if (newUri) {
        /*
         * If policy changed, clear the gear state. Else gear is simply in
         * QUIESCENT state and can be quickly reactivated...
         */
        if (oldPi != pi || oldTclassId.val != tclass->id.val) {
                /*
                 * NewPolicy is TRUE...
                 */
                tmgrMibSwitchedPolicies++;
                tmgrReleaseBwidth(oldTclassId, g);
                tmgrResetGear(g);
        }
        else
                newPolicyOrTclass = FALSE;
}
/*
 * If it is an existing policy, check to see if speed has changed...
 */
if (newPolicyOrTclass == FALSE && g->flags != 0) {
        /*
         * Speed changed ?
         */
        if (g->lastScaledBps != bps)
                tmgrMibSpeedChanges++;
        /*
         * Re-awakening ?
         */
        else if (IS_FLAG_SET(g->flags, GF_QUIESCENT))
                tmgrMibReBursts++;
        /*
         * No change...
         */
        else {
                tmgrMibNoChanges++;
                return OK;
        }
}
/*
 * Else it is a new policy. Shouldn't be any previous allocation
 * unless the in force policy was deleted and this gear changed
 * rate or P-HTTP...
 */
else {
        if (oldPi != pi || oldTclassId.val != tclass->id.val)
                doAccounting = TRUE;
        if (IS_FLAG_SET(g->flags, GF_ACTIVE)){
                tmgrMibSwitchedPolicies++;
                tmgrReleaseBwidth(oldTclassId, g);
                tmgrResetGear(g);
        }
        else
                tmgrMibNewPolicies++;
}

if (newUri) {
        retval = policyCheckBwidth(pi, tclass, g, bps);
}
else {
        policyAllocBwidth(pi, tclass, g, bps);
        retval = OK;
}

return retval;
}

/***************************************************************************
**
 *
 * tclassCheck(flow, tclass, tclassPtr, tspecItemPtr)
 *
 * Check child tclass tspecs and RECURSE until leaf tclass matches.
 * If final traffic class has a policy, apply it. Else search back up
 * through the class tree for an inheritable policy at a matching traffic
 * class.
 *
 * We assume direction is already checked.

***************************************************************************
**
 */
POLICY_ITEM *
tclassCheck(FLOW_SPEC *fspec, TCLASS *tclass, TCLASS **tclassPtr, TSPEC_ITEM
**tiPtr)
{
 POLICY_ITEM      *pi;
 TCLASS           *child;
 whereStr("tclassCheck");
```

```
/*
 * Check Child traffic classes...
 */
for (child = tclass->child; child != NULL; child = child->sibling) {
        /*
         * Do we match this child's tspec(s) ?
         */
        if (tclassMatchTspec(fspec, child->imsTspecList, tiPtr)) {
                /*
                 * We matched. RECURSE to check matching child...
                 */
                blurt2("branching to %s from %s", child->name, tclass->name);
                if ((pi = tclassCheck(fspec, child, tclassPtr, tiPtr)) != NULL)
                        return pi;
        }
}
if (tclass->child != NULL) {
        *tclassPtr = NULL;
        return NULL;
}
/*
 * Is a leaf, belongs to this traffic class. If we have a policy -
 * - apply it. Else search upwards till we find a matching
 * inheritable policy...
 */
*tclassPtr = tclass;

autoRecordFlow(tclass, fspec);

if ((pi = tclass->policy) != NULL) {
        blurt1("policy at tclass %s", tclass->name);
        return(pi);
}
/*
 * No match, find inheritable policy for this flow...
 */
assert((pi = policyFindDefault(tclass, fspec)) != NULL);
return(pi);
}


/*************************************************************************
**
 *
 * tclassMatchTspec(fspec, tclass, tiPtr)
 *
 * Check to see if a flow is a component of a traffic class.
```

```
*****************************************************************************
**
 */
int
tclassMatchTspec(FLOW_SPEC *fspec, TSPEC_ITEM *clist, TSPEC_ITEM **tiPtr)
{
 TSPEC_ITEM *ti;

 *tiPtr = NULL;
 for (ti = clist; ti != NULL; ti = ti->next) {
        if(tspecIsMatchAll(&ti->tspec))
                return(TRUE);

        if(fspec->protFamily != ti->tspec.fspec.protFamily)
                continue;
        if(fspec->protFamily != PF_INET)
                return(TRUE);

        if(ti->tspec.fspec.ipProtocol && fspec->ipProtocol != ti->tspec.fspec.ipProtocol)
                continue;

        if (tspecMatch(fspec, &(ti->tspec))) {
                *tiPtr = ti;
                return(TRUE);
        }
 }
 return(FALSE);
}

/*****************************************************************************
**
 *
 * tspecMatch(FLOW_SPEC *f1, TSPEC *t2)
 *
 * NOTE f1 is the live flow and t2 is the TSPEC.

*****************************************************************************
**
 */
int
tspecMatch(FLOW_SPEC *f1, TSPEC *t2)
{
 FLOW_SPEC *f2 = &(t2->fspec);          /* we treat tspec as a 2nd flow */
 CONN *mask = &(t2->fspecMask.conn);   /* but latch the mask separately */
 int                  prec;

 CONN *c1 = &f1->conn;
 CONN *c2 = &f2->conn;
INT32 hmask;

if(f1->inService && f2->inService && (f1->inService != f2->inService))
        return(FALSE);

if (mask->inside.port && c1->inside.port &&
        ((c1->inside.port < c2->inside.port) || (c1->inside.port > mask->inside.port)))
        return(FALSE);

if (((hmask = mask->inside.host) != 0) &&
        ((c1->inside.host & hmask)  != (c2->inside.host & hmask)))
        return(FALSE);

if(f1->outService && f2->outService && (f1->outService != f2->outService))
        return(FALSE);

if (mask->outside.port && c1->outside.port &&
        ((c1->outside.port < c2->outside.port) || (c1->outside.port > mask->outside.port)))
        return(FALSE);

if (((hmask = mask->outside.host) != 0) &&
        ((c1->outside.host & hmask)  != (c2->outside.host & hmask)))
        return(FALSE);
/*
 * No port spec for ICMP...
 */
if(f1->ipProtocol == ICMP_PTCL
  && (c2->inside.port || c2->outside.port || f2->inService > SVC_CLIENT ||
f2->outService > SVC_CLIENT))
        return(FALSE);
/*
 * IP Precedence is an orthiginal classification element...
 */
prec = PREC(f1->tos);
if(prec < PREC(f2->tos) || prec > PREC(t2->fspecMask.tos))
        return(FALSE);
/*
 * Check for URI here !!!!!! WILDCARD match...
 */
if (t2->uriSpec.uriBufCompile) {
        if(f1->uri == NULL)
                return(FALSE);

        return(httpIsUriMatched(&(t2->uriSpec), f1->uri, f1->uriLen));
}
return(TRUE);
}
```

```
/***************************************************************************
**
 *
 * httpIsUriMatched(uriSpec,stringBuf,stringLen)
 *
 * This matches the URL in stringBuf[] to the
 * (previously-compiled) URI_SPEC *uriSpec.
 *
 * Returns nonzero on match; zero on no-match.

***************************************************************************
**
 */
int
httpIsUriMatched(URI_SPEC *uriSpec,char *stringBuf,int stringLen)
{
 char    *s = stringBuf;
 char    *stringEnd = &stringBuf[stringLen];
 char    *listPtr;
 char    *itemPtr;
 int              itemLen;
 char    *itemEnd;
 int              i;
 char    ich;
 char    sch;
 char    *p, *pEnd;
 char    pch;

 /*
  * Skip over leading "http:"
  */
 if((stringLen >= 6) &&
        (!memcmp(s,"http:",5) ¦¦ !memcmp(s,"HTTP:",5))) {

        stringLen -= 5;
        s += 5;
 }

 listPtr = uriSpec->uriBufCompile->name;

 /*
  * Special case?
  */
 if((uriSpec->flag &
        (URI_FLAG_SPECIAL_EXACT ¦
        URI_FLAG_SPECIAL_LEFT ¦
        URI_FLAG_SPECIAL_RIGHT)) != 0) {

        /*
 * <literal> special-case?  NOTE: This special-case is REQUIRED, as
 * the generic code below does NOT handle the literal-only case
 * correctly.  Doing so would add more code below.
 */
if((uriSpec->flag & URI_FLAG_SPECIAL_EXACT) != 0) {

        /*
         * Length to end of s[] must match exactly.
         */
        if((itemLen = *listPtr++) != stringLen)
                return(0);
}
/*
 * <literal>* special-case?  NOTE: This special-case is correctly
 * handled below; this is a performance hack.
 */
else if((uriSpec->flag & URI_FLAG_SPECIAL_LEFT) != 0) {

        /*
         * s[] must be equal or longer than compare string
         */
        if((itemLen = *listPtr++) > stringLen)
                return(0);
}
/*
 * *<literal> special-case?  NOTE: This special-case is correctly
 * handled below; this is a performance hack.
 */
else if((uriSpec->flag & URI_FLAG_SPECIAL_RIGHT) != 0) {

        ++listPtr;

        /*
         * s[] must be equal or longer than compare string.
         */
        if((itemLen = *listPtr++) > stringLen)
                return(0);

        /*
         * We compare from the right.
         */
        s = (stringEnd - itemLen);
}

/*
 * Case-insensitive compare, stringBuf[] and listPtr[], itemLen bytes
 * (assumes listPtr[] already lower-cased in httpCompileUri())
 */
for(; itemLen != 0; itemLen--) {
```

```
                sch = *s++;
                if(isupper(sch)) sch = tolower(sch);

                if(sch != *listPtr++)
                        return(0);
        }

        /*
         * If we did not fail to match, we must match!
         */
        return(1);
}

/*
 * Loop thru segments
 */
for(i = uriSpec->count; i != 0; i--) {

        /*
         * Latch length and end of this segment.
         * If zero length, matches immediately.  If nonzero, search for
         * match.
         */
        if((itemLen = *listPtr++) != 0) {

                /*
                 * Cannot match if remaining string is shorter than item
                 */
                if(itemLen > (stringEnd - s))
                        return(0);

                /*
                 * Latch beginning and end of this segment.
                 */
                itemPtr = listPtr;
                itemEnd = (itemPtr + itemLen);

                /*
                 * First segment must match exactly on the front.
                 */
                if(i == uriSpec->count) {

                        do {

                                if((ich = *itemPtr) != '?') {

                                        sch = *s;
                                        if(isupper(sch)) sch = tolower(sch);
                                /* if byte doesn't match, NO MATCH */
                                if(ich != sch)
                                        return(0);
                        }

                        s++;

                } while(++itemPtr < itemEnd);
        }
        /*
         * Last segment must match exactly on the end.
         */
        else if(i == 1) {

                s = (stringEnd - itemLen);

                do {

                        if((ich = *itemPtr) != '?') {

                                sch = *s;
                                if(isupper(sch)) sch = tolower(sch);

                                /* if byte doesn't match, NO MATCH */
                                if(ich != sch)
                                        return(0);
                        }

                        s++;

                } while(++itemPtr < itemEnd);
        }
        /*
         * Following segments can accept intervening strings (for '*')
         */
        else {

                /*
                 * End of region where this segment can start
                 */
                pEnd = (stringEnd - itemLen + 1);
again:
                /*
                 * Search for matching first byte (unless it is '?')
                 */
                if((ich = *itemPtr) != '?') {

                        p = s;
                        for(;;) {

                                pch = *p;
                                if(isupper(pch)) pch = tolower(pch);

                                if(ich == pch) {
                                        s = p;
                                        break;
                                }

                                if(!(++p < pEnd))
                                        return(0);
                        }
                }

                /*
                 * Skip over that byte.
                 */
                ++s;
                ++itemPtr;

                /*
                 * Match remaining bytes
                 */
                for(; itemPtr < itemEnd; s++, itemPtr++) {

                        if((ich = *itemPtr) != '?') {

                                sch = *s;
                                if(isupper(sch)) sch = tolower(sch);

                                if(ich != sch) {

                                        /* if out of string, NO MATCH */
                                        if(itemLen > (stringEnd - ++s))
                                                return(0);

                                        /* if not out of string, try again */
                                        itemPtr = listPtr;
                                        goto again;
                                }
                        }
                }
        }

        /*
         * Go on to next segment.
         */
        listPtr = itemPtr;
        }
    }

    /*
     * If we got here, MATCH!
     */
    return(1);
}
```

For building a classification tree by adding a class or adding an additional tspec to an exisitng class and maintaining the specificity order automatically, we have the following...

```
/***************************************************************************
**
 *
 * tclassAddTspec(tclass, tspec)
 *
 * Allocate a TSPEC_ITEM and enlist it at this traffic class.

***************************************************************************
**
 */
staticf void
tclassSetupTspec(TCLASS *tclass, TSPEC_ITEM *ti)
{
 TSPEC_ITEM **lst = &tclass->imsTspecList;

 while (*lst != NULL && tspecLessSpecific(*lst, ti))
        lst = &(*lst)->next;
 ti->next = *lst;
 *lst = ti;

 if (ti == tclass->imsTspecList && ti->next != NULL) {
        tclassRemoveChild(tclass);
        tclassInsertChild(tclass->parent, tclass);
 }
 tcfgConfigChanged = TRUE;
}

/***************************************************************************
**
 *
 * void
 * tclassNew(parent, dir, name, tflags, tspec, tclassPtr)
 *
 *      Make sure TCLASS and BWP direction are the same.

***************************************************************************
**
 */
void
tclassNew(TCLASS *parent,
              int dir,
              char  name,
              int tflags,
                TSPEC *tspec,
                TCLASS **tclassPtr)
{
 whereStr("tclassNew");
 TCLASS              *tclass;
 TSPEC_ITEM          *ti;
 INT32                       dummy;
 BW_POOL             *bwp;
 int                 global = (parent == NULL) ? TRUE : FALSE;

 if ((tclass = tclassAlloc()) == NULL) {
         fatal("Out of TCLASSs!");
 }

 tclass->flags |= (dir & TCF_DIR_MASK) | tflags;

 if (global == FALSE) {
         ti = tclassGetTspec(tclass, tspec);

         VALIDATE_TCLASS(parent);
         tclass->parent = parent;

         if (tclassValidateMembership(parent, tspec, TRUE) != OK) {
                 fatal("TSPEC not reachable");
         }
         bwp = parent->bwp;
 }
 else {
         bwp = GlobalBwPools[dir];
         tclass->flags |= TCF_GLOBAL+TCF_CIRLIMIT_NONE;
 }
 /*
  * Thread the tclass into its bandwidth pool as well...
  */
 tclass->bwp = bwp;
 bwpAddTclass(bwp, tclass);
 /*
  * If new tclass is NOT global, thread it into the classifier tree.
  * It should be inserted in Match Specificity order onto the
  * child.sibling list.
  */
 if (global == FALSE) {
         tclassSetupTspec(tclass, ti);
         tclassInsertChild(parent, tclass);
         bwpRedistributeBwidth(dir);
 }
 *tclassPtr = tclass;
}
```

```
/*************************************************************************
**
 *
 * tclassInsertChild(parent, tclass)
 *
 * Insert in child list in DMS order.

*************************************************************************
**
 */
void
tclassInsertChild(TCLASS *parent, TCLASS *tclass)
{
 TCLASS *sibling, *last;
 unsigned long old, dummy;

 sibling = parent->child;
 last = NULL;

 LOCK_TASK(&old);
 while(sibling != NULL && tclassLessSpecific(tclass, sibling)){
        last = sibling;
        sibling = sibling->sibling;
 }
 tclass->sibling = sibling;
 tclass->parent = parent;

 if (last == NULL)
        parent->child = tclass;
 else
        last->sibling = tclass;

 UNLOCK_TASK(old,&dummy);

 tcfgConfigChanged = TRUE;
}

/*************************************************************************
**
 *
 * tclassLessSpecific(a, b)
 *
 * Return TRUE if a is less match specific than b, that is if a should
 * be ordered AFTER b...

*************************************************************************

  /
staticf int
```

```
tclassLessSpecific(TCLASS *a, TCLASS *b)
{
 int rc;

 if (IS_FLAG_SET(a->flags, TCF_EXCEPTION)){
        if (! IS_FLAG_SET(b->flags, TCF_EXCEPTION))
               return(FALSE);
 }
 else if (IS_FLAG_SET(b->flags, TCF_EXCEPTION))
        return(TRUE);

 if (IS_FLAG_SET(a->flags, TCF_INHERITABLE)){
        if (! IS_FLAG_SET(b->flags, TCF_INHERITABLE))
               return(TRUE);
 }
 else if (IS_FLAG_SET(b->flags, TCF_INHERITABLE))
        return(FALSE);
 /*
  * Ok - need to order by tspec IMS...
  */
 if ((rc = tspecLessSpecific(a->imsTspecList, b->imsTspecList)) == TRUE
               && tclassIsMatchAll(b) && a->child != NULL) {
        if (b->child != NULL)
               rc = tclassLessSpecific(a->child, b->child);
        rc = FALSE;
 }
 /*
  * Ok - now order by policy/no policy...
  */
 if (rc == TRUE && a->policy == NULL && b->policy != NULL)
        return(FALSE);
 return(rc);
}


/*************************************************************************
**
 *
 * tspecLessSpecific(a, b)
 *
 * Return TRUE if (a) less specific than (b).

*************************************************************************
**
 */
#define TSPEC_EQ(a,b) (memcmp((a), (b), sizeof(TSPEC)) == 0)

int
tspecLessSpecific(TSPEC_ITEM *a, TSPEC_ITEM *b)
```

```
{
 int abits, bbits;
 FLOW_SPEC *amask, *bmask, *af, *bf;
 whereStr("tspecLessSpecific");

 if(tspecIsMatchAll(&a->tspec))
        return(TRUE);
 else if(tspecIsMatchAll(&b->tspec))
        return(FALSE);

 af = &a->tspec.fspec;
 bf = &b->tspec.fspec;

 if(af->protFamily != bf->protFamily)
        return(af->protFamily > bf->protFamily);

 if(af->protFamily != PF_INET)
        return(FALSE);

 /*
  * OK - Compare host masks...
  */

 amask = &a->tspec.fspecMask;
 bmask = &b->tspec.fspecMask;

 abits = tspecHostBits(&amask->conn);
 bbits = tspecHostBits(&bmask->conn);

 if (abits != bbits) return(abits < bbits);
 /*
  * Compare port ranges...
  */
 abits = tspecPortBits(&af->conn, &amask->conn);
 bbits = tspecPortBits(&bf->conn, &bmask->conn);

 if (abits != bbits) return(abits > bbits);                    /* here more "bits" mean less
specific */

 abits = (af->service[DIR_INBOUND] > SVC_CLIENT);
 bbits = (bf->service[DIR_INBOUND] > SVC_CLIENT);

 if(abits != bbits)
        return(abits < bbits);

 abits = (af->service[DIR_OUTBOUND] > SVC_CLIENT);
 bbits = (bf->service[DIR_OUTBOUND] > SVC_CLIENT);

 if(abits != bbits)
```

```
        return(abits < bbits);

/*
 * Compare precedence ranges
 */
abits = PREC(amask->tos) - PREC(af->tos);
bbits = PREC(bmask->tos) - PREC(bf->tos);

if(abits != bbits)
        return(abits > bbits);

/*
 *  Finally - compare URL if it is present in either...
 */
if (af->uri || bf->uri)  return(urlLessSpecific(af->uri, bf->uri));
/*
 * They are the same - true on a tie...
 */
return(TRUE);
}

/***************************************************************************
**
 *
 * urlLessSpecific(a, b)
 *

***************************************************************************
**
 */
int
urlLessSpecific(char *a, char *b)
{
 int aw, bw;
 char *cp;

        if (! a)
                return(TRUE);
        else if (!b)
                return(FALSE);
        /*
         * OK - both have URL specified. Compare number of wildcards...
         */
 for (aw = 0, cp = a; *cp; cp++){
        if (*cp == '*') aw++;
 }
 for (bw = 0, cp = b; *cp; cp++){
        if ( cp == '*') bw++;
 }
 return(aw > bw ? TRUE : FALSE);
}


int
hostBits(INT32 host)
{
 int bits, i;

 for (bits = 0, i = 31; i >= 0; i--){
        if ((1 << i) & host)
                bits++;
        else
                break;
 }
 return(bits);
}

int
portBits(INT16 port, INT16 range)
{
 if (port == 0)
        return(0x00007fff);
 else if (range == 0)
        return(1);
 else
        return((range-port)+1);
}

int
tspecHostBits(CONN *conn)
{
 return(hostBits(conn->inside.host) + hostBits(conn->outside.host));
}

int
tspecPortBits(CONN *conn, CONN *mask)
{
 return(portBits(conn->inside.port, mask->inside.port) +
                portBits(conn->outside.port, mask->outside.port));
}


/***************************************************************************
**
 *
 * tclassValidateMembership(parent, tspec, verbose)

 * Search recursively through parents to make sure we have a path from the
 * root down to here. We ignore siblings and uncles, as specificity-order
 * should take care of that.

****************************************************************************
**
 */
int
tclassValidateMembership(TCLASS *parent, TSPEC *tspec, int verbose)
{
 VALIDATE_TCLASS(parent);
 /*
  * If we got to the top - it is valid...
  */
 if (IS_FLAG_SET(parent->flags,TCF_GLOBAL)) return(OK);
 /*
  * Is tspec a child of this parent...
  */
 if (! tspecIsChild(tspec, parent->imsTspecList)){
        if (verbose) printf("\nNot valid member of traffic class %s", parent->name);
        return(NOTOK);
 }
 /*
  * OK - we fit here. Check at next parent level...
  */
 return(tclassValidateMembership(parent->parent, tspec, verbose));
}
```

Speed Scaling -

when a flow is allocated, the GIR and EIR are scaled per the detected (potential maximum) speed.

in this embodiment, the speed scaling is done by extrapolating from a defined high speed and low speed target rates.

```
/****************************************************************************
**

 * file: scale.c
 *
 * description: CIR and EIR scaling to load.
 *
 *
 *              SCALING CIR and EIR to load potential:
 *
 *                      CIR scale is specified by user as two values, a base value
 *                      (B) under which all speeds get 100%, and a limiting value (L)
 *                      to cap CIR for high speed users. If a connections incoming
 *                      speed is actually BETWEEN (B) and (L), then we create a
 *                      scaled intermediate value for CIR.
 *
 *                      So CIR_SPEC = {B,L}
 *
 *                      EIR scale is used to generate an EIR_VECTOR of allocation limits
 *                      at each EIR priority level. EIR scale is specified as the
 *                      highest priority level (HP), the scaled limit at that priority
 *                      level (in terms of B and L, as with CIR). We then will
automatically
 *                      generate the rest of the vector for the lower priority levels, up
 *                      to an optional total limit (TL).
 *
 *                      So EIR_SPEC = {HP, HP-B, HP-L, TL};
 *
 * 09/18/97 Copyright Packeteer, Inc, 1996-97 @(#)scale.c        1.2

*******************************************************************************
**
 */


/*
 * Rate Specifier, i.e. CIR + EIR...
 */
typedef struct {
 BPS                cirLimit;           /* How much CIR I want */
 BPS                eirDemand;            /* total of eirDemandvector */

 EIR_VECTOR         eirDemandVector;        /* How much EIR I want */
 /*
  * UDP stuff tbd...
  */
 INT32                      bucketLimit;
} RATE_SPEC;


typedef struct _gearshift {
 DLINK                      dlink;
#define GF_INITIALIZED              0x0001
#define GF_SCALED                          0x0002
#define GF_ACTIVE                          0x0004
#define GF_EIR_OVERALLOCATED        0x0008       /* allocated though none
available*/
#define GF_CIR_OVERALLOCATED        0x0010       /* allocated though none
available*/
#define GF_QUIESCENT                       0x0020
#define GF_PRIORITY                        0x0040
#define GF_RATE                            0x0080
#define GF_DQ                                     0x0100
#define GF_TCPRC                           0x0200
#define GF_FILTER                          0x0400
#define GF_NEEDS_REALLOC            0x0800
#define GF_CIR_COMMITTED            0x1000

 INT32                    flags;
 BPS               target;
 INT32                    cirInUse;
 INT32                    eirInUse;
 BPS               lastScaledBps;
 RATE_SPEC rspec;
} GEARSHIFT;

/*
 * For scaling CIR to connection speed. See traffic/scale.c.
 * If limit is set to 0, no CIR is allocated.
 */
typedef struct {
 BPS base;
 BPS limit;
} CIR_SPEC;

/*
 * For scaling EIR to connection speed, i.e. creating eirLimits vector.
 * hp -highest priority, hp-base, and hp-limit and optional total limit.
 * See traffic/scale.c.
 *
 * As with CIR Spec, if hpLimit is set to 0, no EIR is allocated.
 */
typedef struct {
 INT8 hp;
 BPS  hpBase;
 BPS  hpLimit;
 BPS  totalLimit;
} EIR_SPEC;


/*
```

```
 * Policy Item
 */
typedef struct _policy {
 struct _policy          *next;
 struct _tclass  *tclass;

 INT32                   flags;
 INT8                    direction;
 INT8                    sessionType;

 INT32                           hits;
 /*
  * For scaling EIR/CIR...
  */
 EIR_SPEC                eirSpec;
 CIR_SPEC                cirSpec;
 INT32                           bucketLimit;
 /*
  * For Admission Control...
  */
 URI_BUF                 *redirectUrl;
 /*
  * For Content Substitution...
  */
 SUB_SPEC    *subSpec;
} POLICY_ITEM;


/*************************************************************************
**
 *
 * scaleToLoad(policy, tclass, gears, bps)
 *
 * Set up EIR limit vector as well as appropriate CIR for this specific
 * connection speed (bps) for this GEAR (g).

*************************************************************************
**
 */
void
scaleToLoad(POLICY_ITEM *pi, TCLASS *tclass, GEARSHIFT *g, SPEED bps)
{
 CIR_SPEC *cs = &pi->cirSpec;
 EIR_SPEC *es = &pi->eirSpec;
 BW_POOL     *bwp = tclass->bwp;
 int             eirDemand, eirLimit, eirExtra;

 g->lastScaledBps = bps;
/*
 * First set CIR...
 */
if (cs->limit > 0) {
        g->rspec.cirLimit = scaleRate(bps,cs->base,cs->limit, tclass->bwp->limit);
}
else
        g->rspec.cirLimit = g->cirInUse = 0;

eirDemand = bps - g->rspec.cirLimit;

eirVectorSet(g->rspec.eirDemandVector, 0); /* Zero vector... */
g->rspec.eirDemand  = 0;

if (es->hpLimit > 0 && eirDemand > 0) {
        int     cap;
        /*
         * Descending priority levels will get exponential chunks of
         * bandwidth, increasing in size as priority decreases.
         */
        eirLimit = scaleRate(eirDemand, es->hpBase, es->hpLimit, bwp->limit);
        /*
         * We allocate all of the rest (of BPS) at lower priority
         * levels unless a TOTAL LIMIT is set...
         */
        cap = bwp->limit;
        cap -= g->rspec.cirLimit;

        if (es->totalLimit > 0 && es->totalLimit < cap) cap = es->totalLimit;
        /*
         * cap is now the largest amount of EIR we can have
         */
        eirExtra = min(eirDemand, cap) - eirLimit;
        if (eirExtra < 0) eirExtra = 0;

        /*
         * Assign limits to descending priority levels...
         */
        (void)eirAssignDemand(es->hp, eirLimit, eirExtra, &g->rspec);
}
info4("pol %s: gear %d cir=%d, eir=%d",
          pi->tclass->name, g, g->rspec.cirLimit, g->rspec.eirDemand);
}


/* *********************************************** *****************************
*
 *
 *         scaleRate(bps, base, limit, total)
 *

*************************************************************************
**
 */
INT32
scaleRate(INT32 bps, INT32 base, INT32 limit, INT32 total)
{
 INT32          pct;
 /*
  * If detected rate is lower than scale-base, just return base.
  */
 if (bps <= base) {
        bps <<= 2;
        if (bps > base) bps = base;
        return bps;
 }
 /*
  * Constrain total...
  */
 if (total > limit << 1) total = limit << 1;
 /*
  * If detected rate higher than total, just return limit
  */
 if (bps >= total)
        return limit;
 /*
  * OK - we will allocate base + some percentage of (limit - base),
  * i.e. up to limit. Since we know bps < total, this will always
  * return a number less than or equal to limit.
  */
 pct = (bps << 7)/ total;
 return base + (((limit - base) * pct) >> 7);
}

/************************************************************************
**
 *
 * eirAssignDemand(hp, hpLimit, eirExtra, rspec)
 *

*************************************************************************
**
 */
staticf void
eirAssignDemand(int hp, int eirLimit, int eirExtra, RATE_SPEC *r)
{
 BPS    *b = &r->eirDemandVector[hp];
 int     i, active;
 int     extra, extraChunk;
 int     left;

 extra = max(0, (eirExtra - (eirLimit*(hp+1))));
 left = eirExtra + eirLimit;

 r->eirDemand = 0;
 active = 0;
 /*
  * NOTE: Eir demand has been set to 0...
  */
 for (i = hp; i >= 0; i--, b--) {

         if (left < EIR_MIN_CHUNK && active != 0)
                 break;

         active = 1;

         if (i == 0)
                 *b = left;
         else {
                 *b = min(left, eirLimit);

                 if (extra > 0 &&  i < hp) {
                         extraChunk = extra >> (i + 1);
                         *b += extraChunk;
                 }
                 left -= *b;
         }
         r->eirDemand += *b;
 }
}
```

WHAT IS CLAIMED IS:

1. A method for allocating bandwidth in a packet telecommunication system having any number of flows, including zero, using a classification paradigm comprising the steps of:

parsing a packet into a flow specification, wherein said flow specification contains one or more instances of any of the following:

a protocol family designation,

a direction of packet flow designation,

a protocol type designation,

a plurality of hosts,

a plurality of ports,

in http protocol packets, a pointer to a URL; thereupon,

matching the flow specification of the parsing step to a plurality of hierarchically-recognized classes represented by a plurality of nodes, each node having a traffic specification and a mask, according to the mask; thereupon,

having found a matching node in the matching step, associating said flow specification with one of said plurality of hierarchically-recognized classes represented by a plurality nodes; and

allocating bandwidth resources according to a policy associated with said class.

2. The method of claim 1 wherein a node may have one or many traffic specifications.

3. The method of claim 1 wherein said plurality of hierarchically-recognized classes represents a plurality of orthogonal flow attributes.

4. The method of claim 1 wherein allocating bandwidth resources further comprises the steps of:

for rate based policies having a guaranteed information rate (GIR) and an excess information rate (EIR), allocating GIR until no further GIR is available; thereupon, allocating EIR,

for priority based policies, having a priority, allocating unreserved service according to said priority.

5. The method of claim 1 wherein said allocating bandwidth resources further comprises the step of:

allocating a combination of GIR resources, EIR resources and unreserved resources, wherein GIR resources and EIR resources are allocated based upon speed scaling and unreserved resources are allocated based upon a priority.

6. The method of claim 5 wherein allocating guaranteed information rate (GIR) resources and excess information rate (EIR) resources to said flow based upon speed scaling further comprises the steps of:

detecting the speed of a network link upon which said flow is transported;

allocating available bandwidth resources based upon the speed detected.

7. The method of claim 5 wherein allocating unreserved resources to a particular flow, said flow having an input rate, said flow being one of said plurality of flows, based upon a priority further comprises the steps of:

extrapolating from said input rate of the detecting step an extrapolated unreserved demand (EUD);

aggregating together a plurality of individual flow demands to form an aggregate rate demand (ARD);

combining said EUD and said ARD to form an instantaneous total demand;

allocating available bandwidth resources based upon the instantaneous total demand determined in the combining step from highest priority demand first to lower priority demanded to yield a satisfaction percentage of demand;

determining a target rate for individual flows by multiplying said individual flow demand for each flow by said satisfaction percentage of demand determined in the allocating step.

8. The method of claim 7 wherein said unreserved resources are comprised of a plurality of individual inputs, each input having a priority, further comprising the step of:

determining a target rate for said units according to said priority by multiplying said extrapolated unreserved demand by said satisfaction percentage of demand determined in the allocating step.

9. The method of claim 1 further comprising the step of determining that if guaranteed information rate resources are insufficient to accommodate said flow, invoking an admissions policy.

10. In a packet communication environment allocated into layers, including at least a transport layer, link layer and an application layer, a method for classifying Internet Protocol (IP) flows at the transport layer for use in determining a policy, or rule of assignment of a service level, and enforcing that policy by direct rate control, said method comprising:

applying individual instances of traffic classes to a classification tree based on selectable information obtained from an application layer of a communication protocol; and thereupon,

mapping the IP flow to the defined traffic classes, said traffic classes having an associated policy.

11. A method for managing bandwidth on Internet Protocol (IP) flows in a packet communication environment allocated into layers, including at least a transport layer, a link layer and an application layer, said method comprising:

automatically detecting selectable information about said flow;

determining a policy for assigning a service level to said flow based upon said automatically detected selectable information about said flow; and

implementing said policy by direct rate control of said flow.

12. The method of claim 11 wherein said determining step further comprises applying individual instances of traffic class specifications, each traffic class specification having an associated policy, to said flow based on selectable information; thereupon,

associating with said flow the particular policy associated with said traffic class.

13. The method of claim 12 wherein said traffic classes have a hierarchical relationship to one another.

14. A method for allocating network bandwidth, which bandwidth comprises guaranteed information rate (GIR) resources, excess information rate (EIR) resources and unreserved rate resources, on Internet Protocol (IP) flows in a packet communication environment allocated into layers, including at least a transport layer, a link layer and an application layer, said method comprising:

automatically detecting selectable information about said flow;

determining a policy for assigning a service level to said flow based upon said automatically detected selectable information about said flow, said policy directing either reserved service or unreserved service;

allocating bandwidth representing a combination of GIR resources, EIR resources and unreserved resources to a plurality of flows assigned said service level by said determining step.

15. The method of claim 14 wherein said unreserved resources are comprised of a plurality of individual inputs, each input having a priority, further comprising the step of:

determining a target rate for said units according to said priority by multiplying said extrapolated unreserved demand by said satisfaction percentage of demand determined in the allocating step.

16. The method of claim 14 wherein said allocating bandwidth representing a combination of GIR resources, EIR resources and unreserved resources comprises allocating GIR resources and EIR resources based upon a speed scaling and allocating unreserved resources based upon a priority.

17. The method of claim 16 wherein allocating unreserved resources based upon a priority to a plurality of flows, each having an input rate, further comprises the steps of:

detecting the input rate of a flow;

extrapolating from said input rate of the detecting step an extrapolated unreserved demand (EUD);

aggregating together a plurality of individual flow demands to form an aggregate rate demand (ARD);

combining said EUD and said ARD to form an instantaneous total demand;

allocating available bandwidth resources based upon the instantaneous total demand determined in the combining step from highest priority demand first to lower priority demanded to yield a satisfaction percentage of demand;

determining a target rate for individual flows by multiplying said individual flow demand for each flow by said satisfaction percentage of demand determined in the allocating step.

18. The method of claim 17 wherein said unreserved resources are comprised of a plurality of individual inputs, each input having a priority, further comprising the step of:

determining a target rate for said units according to said priority by multiplying said extrapolated unreserved demand by said satisfaction percentage of demand determined in the allocating step.

19. The method of claim 14 further comprising the step of determining that if guaranteed bandwidth available is insufficient to accommodate said flow, invoking an admissions policy.

20. The method of claim 14 wherein said determining step further comprises applying individual instances of traffic classes, each traffic class having an associated policy, to said flow based on selectable information; thereupon,

associating with said flow the particular policy associated with said traffic class.

21. The method of claim 19 wherein said traffic classes have a hierarchical relationship to one another.

22. The method of claim 14 further comprising allocating said bandwidth representing unreserved data resources among flows based upon a priority associated with said flows.

23. The method of claim 14 further comprising applying a partition to one or more classes, said partition having a limit on aggregate data rate flow.

24. The method of claim 23 wherein said limit is a soft partition limiting FIR for one or more classes.

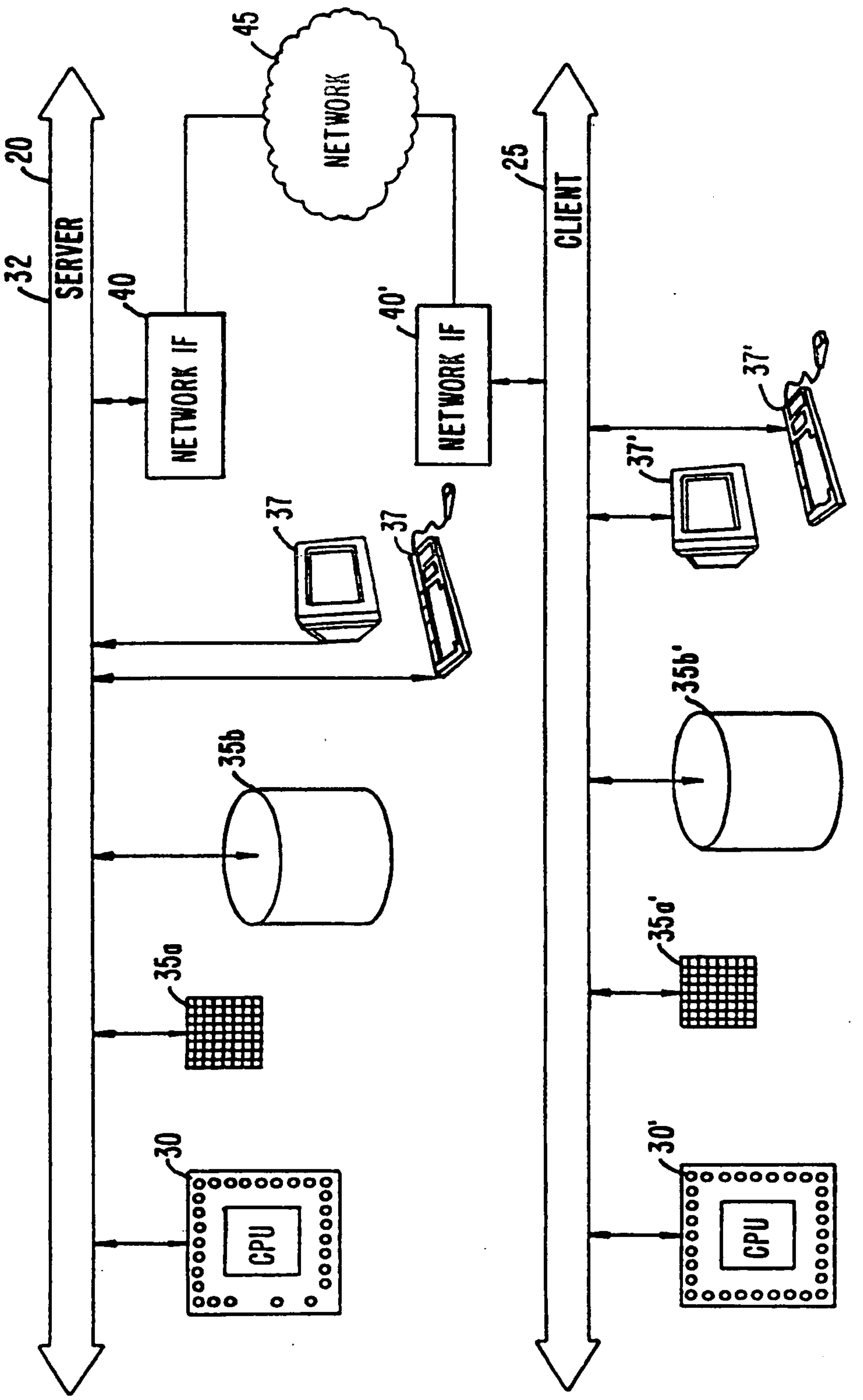
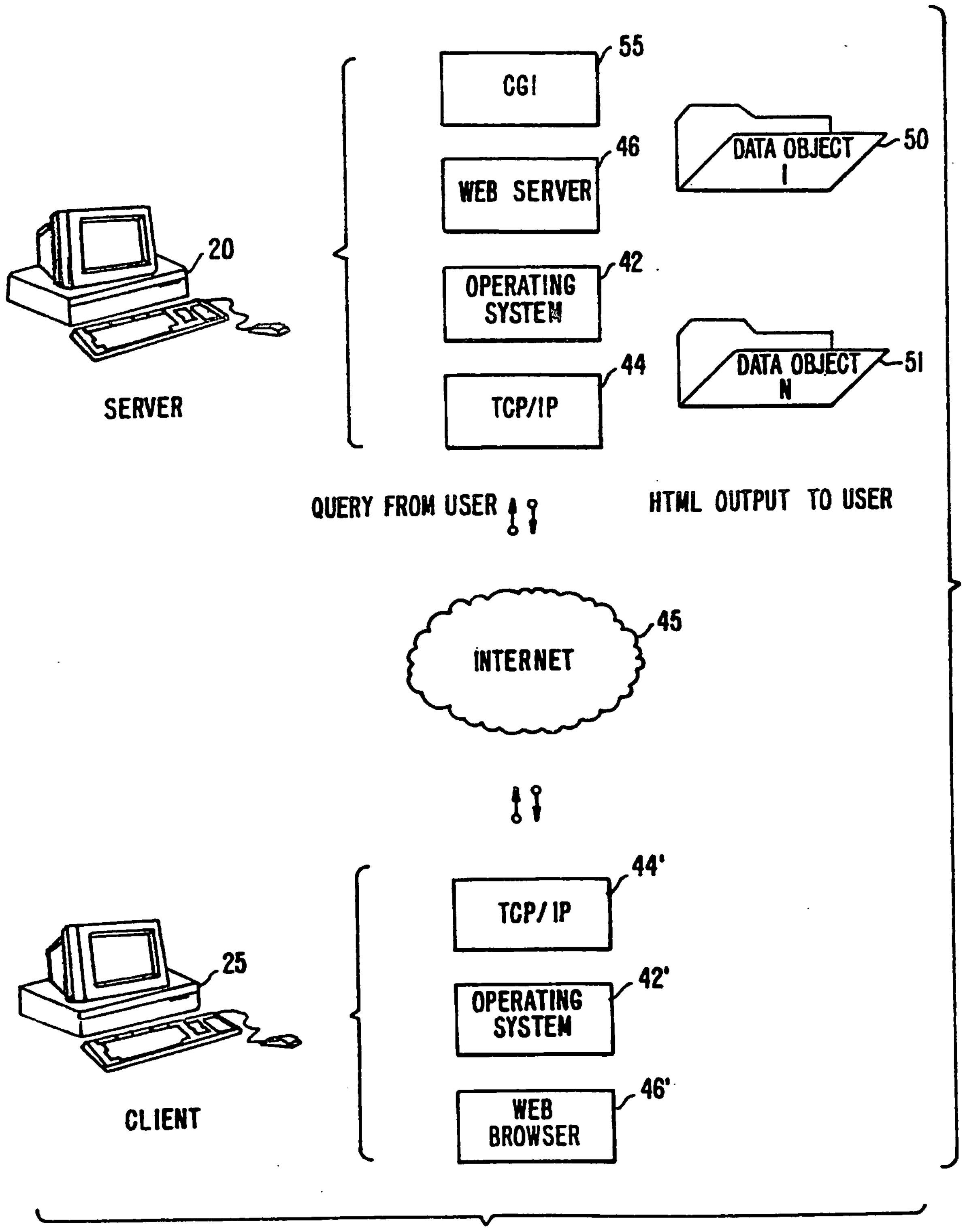


FIG. 1A.
(PRIOR ART)



CGI
55
WEB SERVER
46
OPERATING SYSTEM
42
TCP/IP
44
DATA OBJECT 1
50
DATA OBJECT N
51
20
SERVER
QUERY FROM USER
HTML OUTPUT TO USER
INTERNET
45
TCP/IP
44'
OPERATING SYSTEM
42'
WEB BROWSER
46'
25
CLIENT

FIG. 1B.

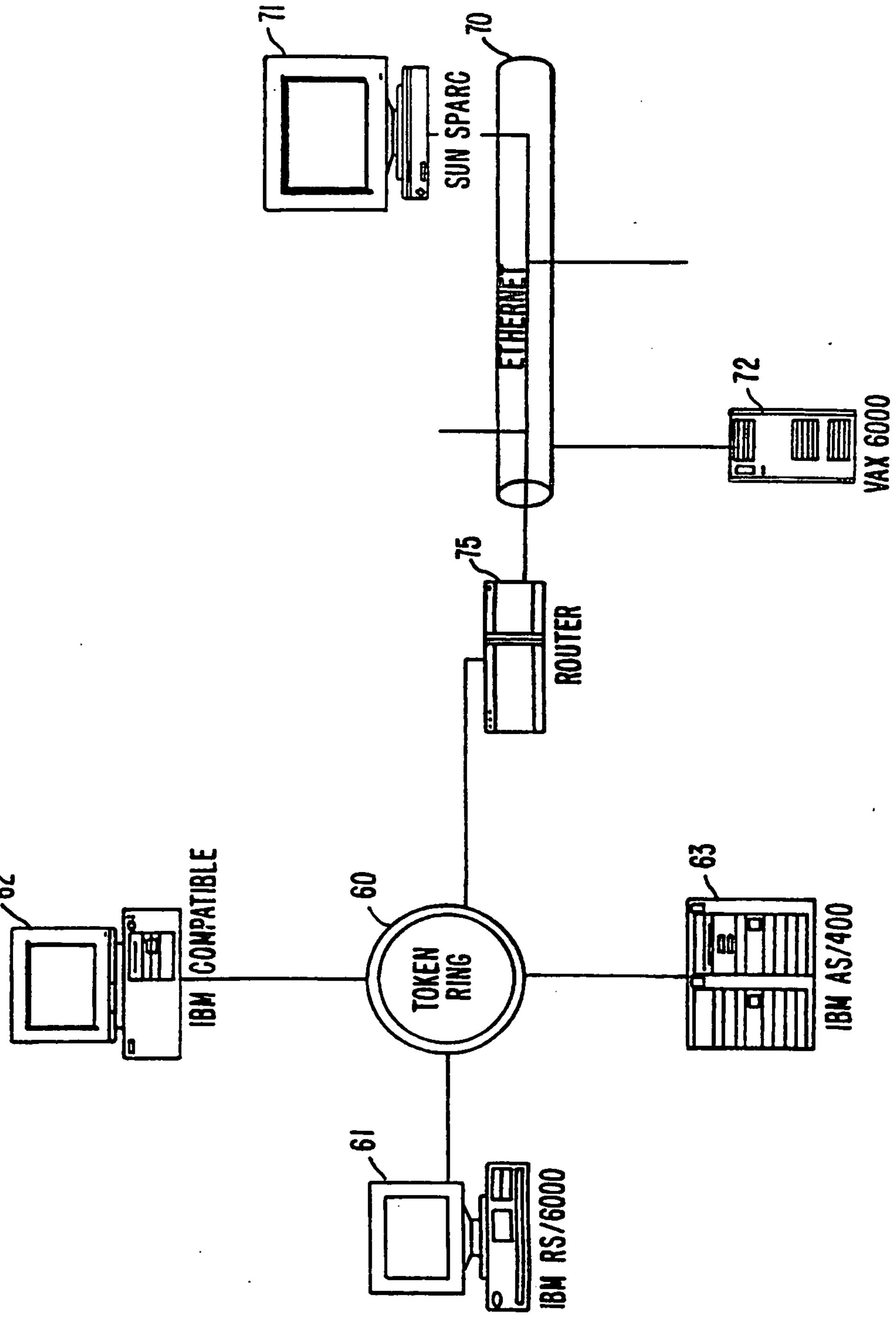
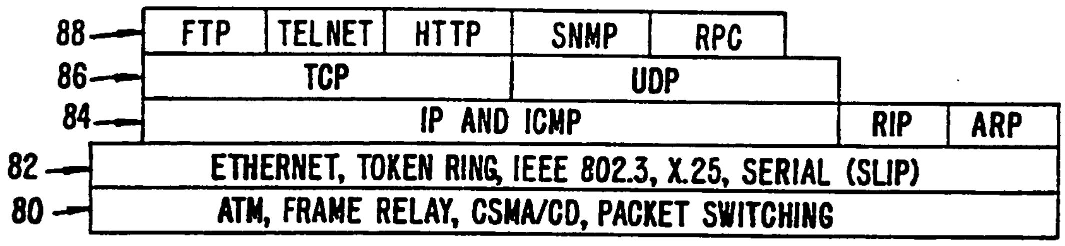


FIG. 1C. (PRIOR ART)


88
FTP
TELNET
HTTP
SNMP
RPC
86
TCP
UDP
84
IP AND ICMP
RIP
ARP
82
ETHERNET, TOKEN RING, IEEE 802.3, X.25, SERIAL (SLIP)
80
ATM, FRAME RELAY, CSMA/CD, PACKET SWITCHING

LEGEND	
88	SESSION/APPLICATION LAYER
86	TRANSPORT LAYER
84	NETWORK LAYER
82	DATA LINK LAYER
80	PHYSICAL LAYER

FIG. 1D.

(PRIOR ART)

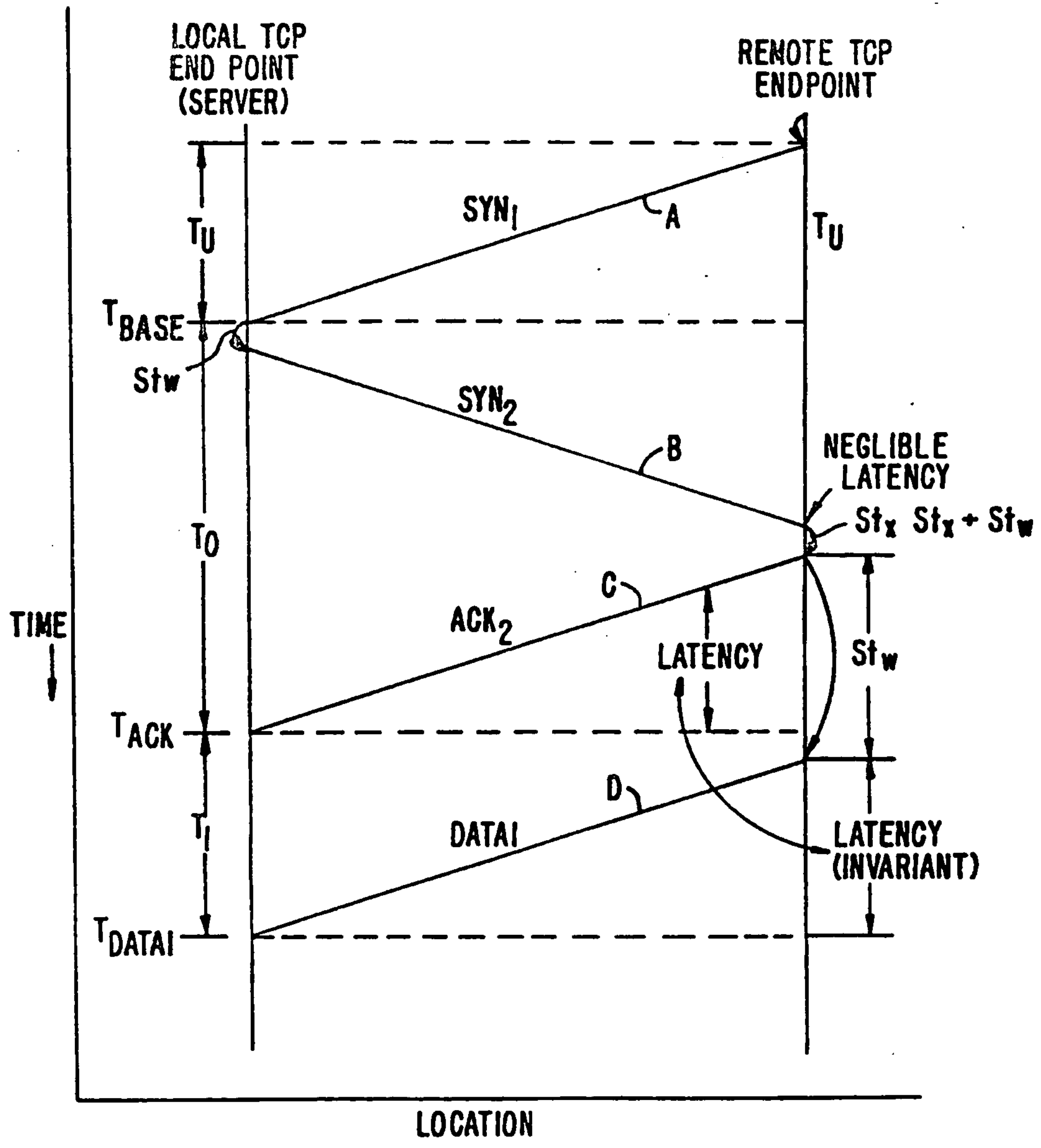


FIG. 1E.

(PRIOR ART)

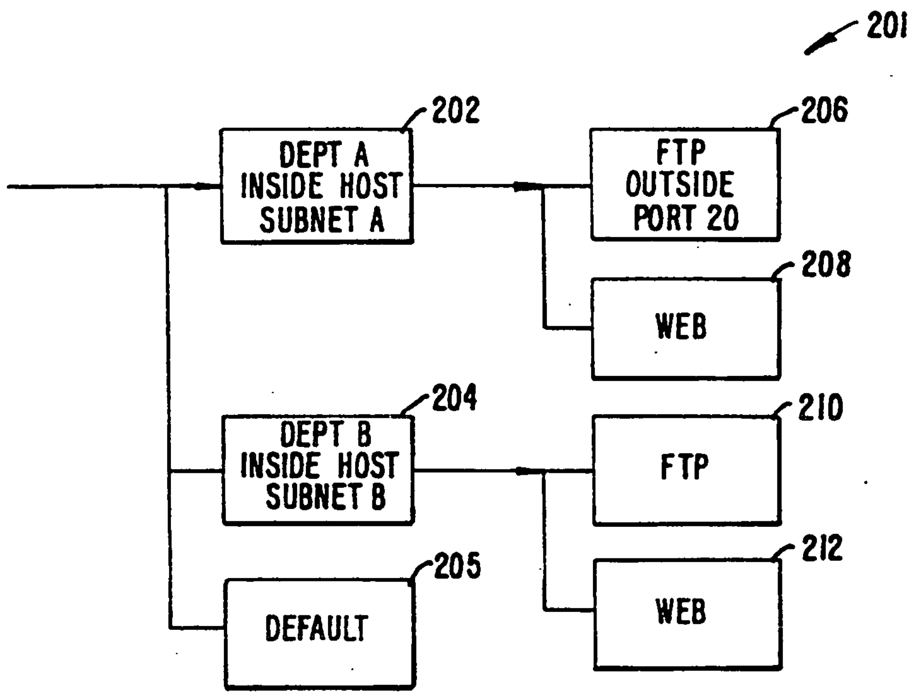

201
202
DEPT A INSIDE HOST SUBNET A
206
FTP OUTSIDE PORT 20
208
WEB
204
DEPT B INSIDE HOST SUBNET B
210
FTP
205
DEFAULT
212
WEB

FIG. 2A.

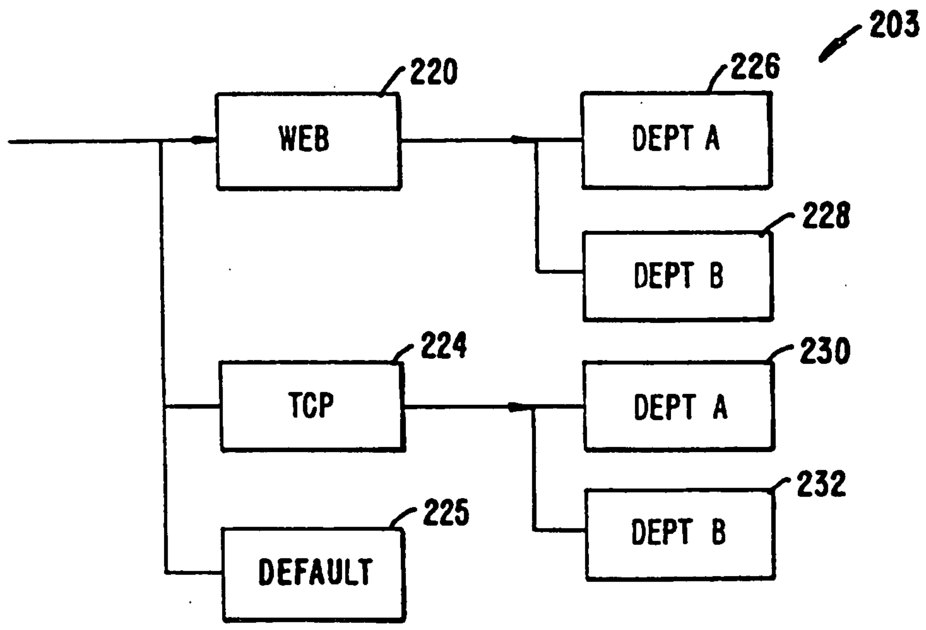

203
220
WEB
226
DEPT A
228
DEPT B
224
TCP
230
DEPT A
225
DEFAULT
232
DEPT B

FIG. 2B.

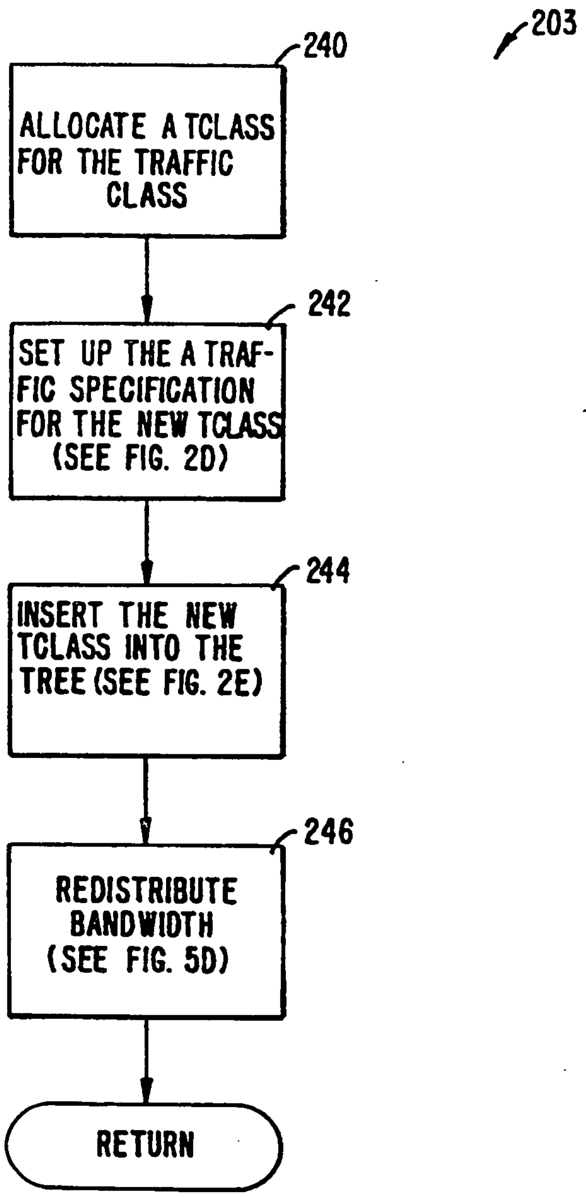

203
240
ALLOCATE A TCLASS FOR THE TRAFFIC CLASS
242
SET UP THE A TRAFFIC SPECIFICATION FOR THE NEW TCLASS (SEE FIG. 2D)
244
INSERT THE NEW TCLASS INTO THE TREE (SEE FIG. 2E)
246
REDISTRIBUTE BANDWIDTH (SEE FIG. 5D)
RETURN

FIG. 2C.

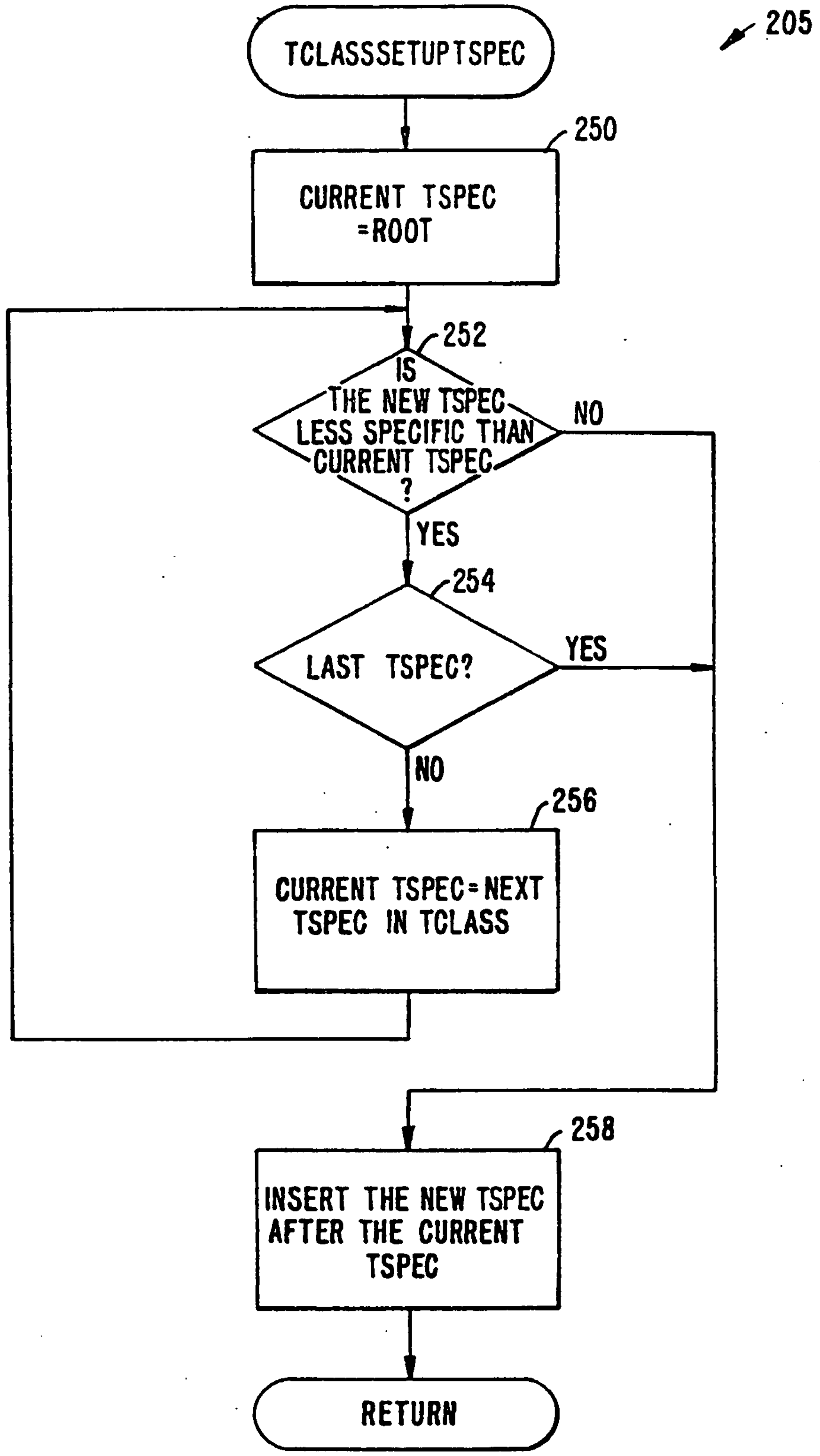

205
TCLASSSETUPTSPEC
250
CURRENT TSPEC =ROOT
252
IS THE NEW TSPEC LESS SPECIFIC THAN CURRENT TSPEC ?
NO
YES
254
LAST TSPEC?
YES
NO
256
CURRENT TSPEC=NEXT TSPEC IN TCLASS
258
INSERT THE NEW TSPEC AFTER THE CURRENT TSPEC
RETURN

FIG. 2D.

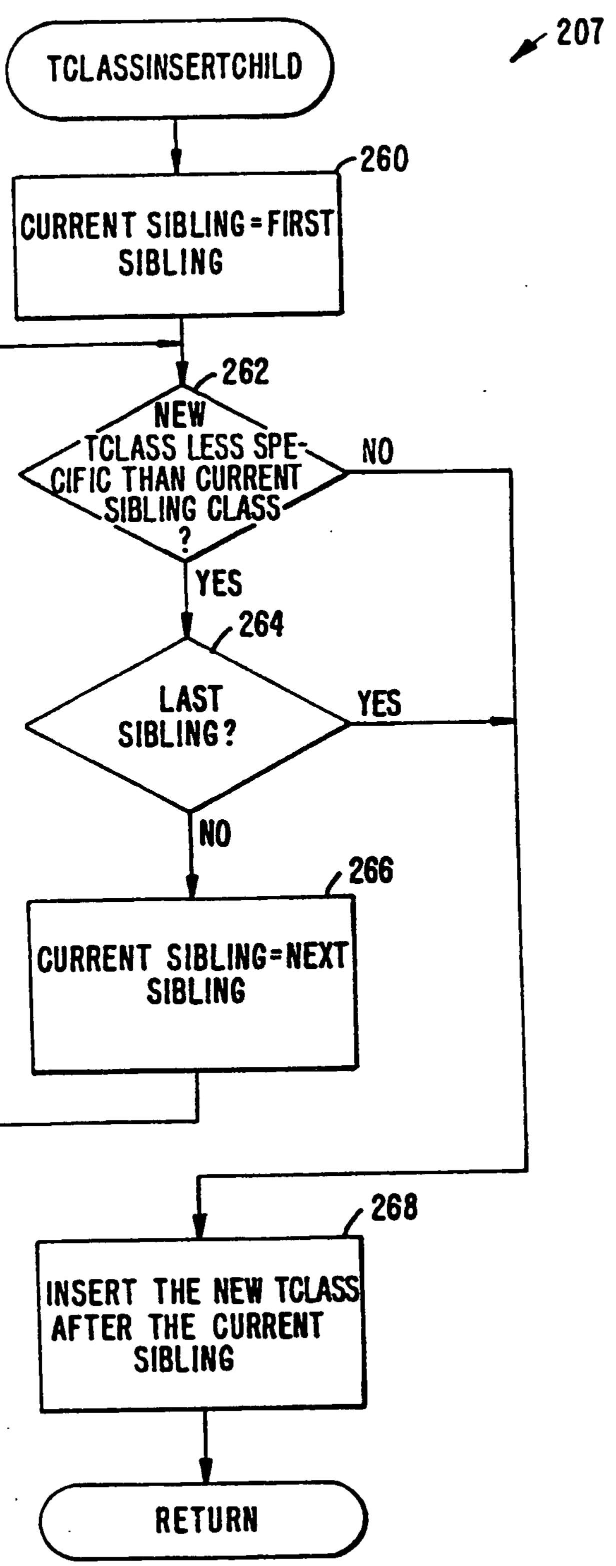

207
TCLASSINSERTCHILD
260
CURRENT SIBLING=FIRST SIBLING
262
NEW TCLASS LESS SPECIFIC THAN CURRENT SIBLING CLASS ?
NO
YES
264
LAST SIBLING?
YES
NO
266
CURRENT SIBLING=NEXT SIBLING
268
INSERT THE NEW TCLASS AFTER THE CURRENT SIBLING
RETURN

FIG. 2E.

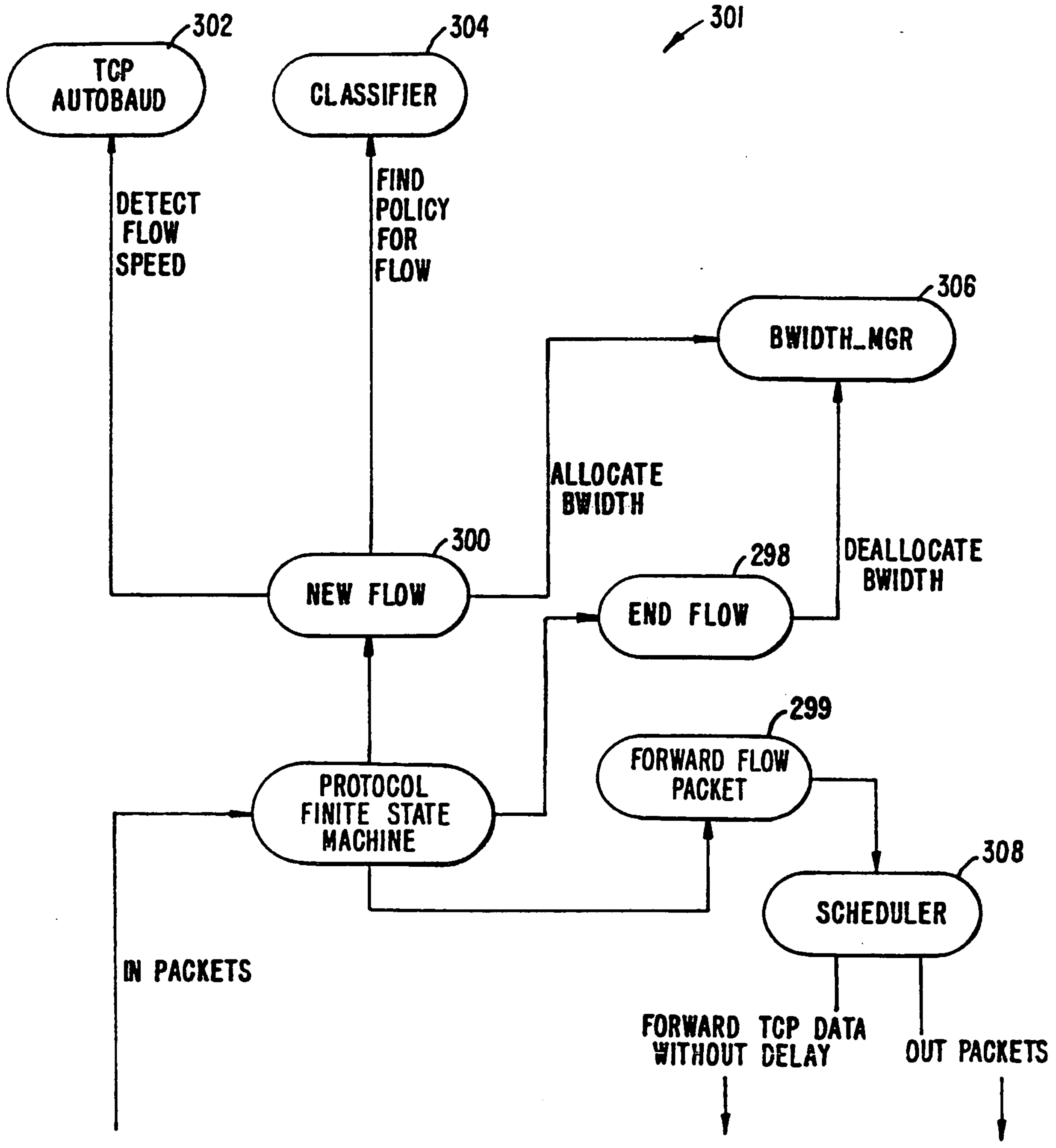
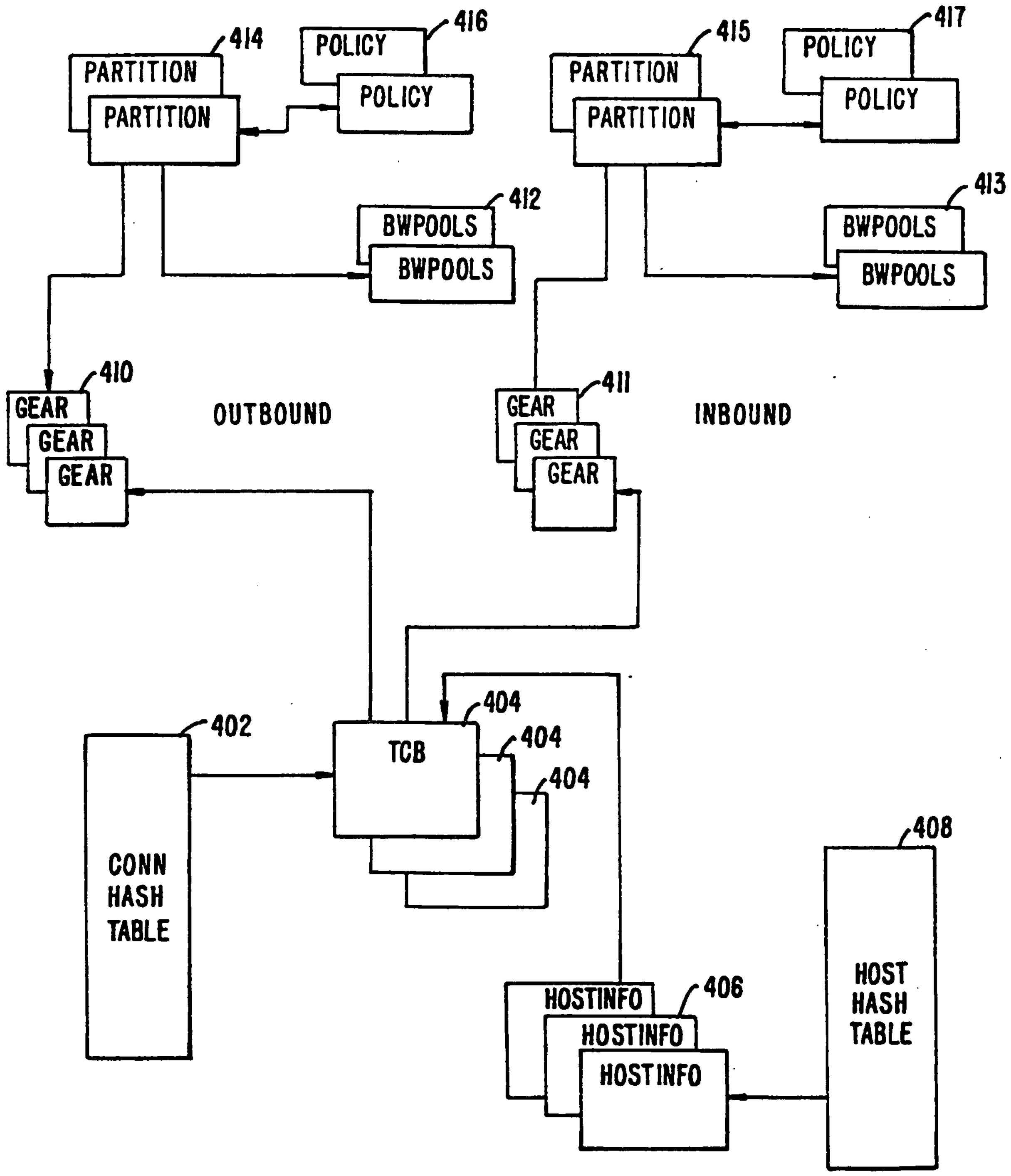

301
302
TCP AUTOBAUD
304
CLASSIFIER
DETECT FLOW SPEED
FIND POLICY FOR FLOW
306
BWIDTH_MGR
ALLOCATE BWIDTH
300
NEW FLOW
298
END FLOW
DEALLOCATE BWIDTH
299
FORWARD FLOW PACKET
PROTOCOL FINITE STATE MACHINE
308
SCHEDULER
IN PACKETS
FORWARD TCP DATA WITHOUT DELAY
OUT PACKETS

FIG. 3.



PARTITION
PARTITION
414
POLICY
POLICY
416
BWPOOLS
BWPOOLS
412
GEAR
GEAR
GEAR
410
OUTBOUND
PARTITION
PARTITION
415
POLICY
POLICY
417
BWPOOLS
BWPOOLS
413
GEAR
GEAR
GEAR
411
INBOUND
CONN HASH TABLE
402
TCB
404
404
404
HOSTINFO
HOSTINFO
HOSTINFO
406
HOST HASH TABLE
408

FIG. 4A.

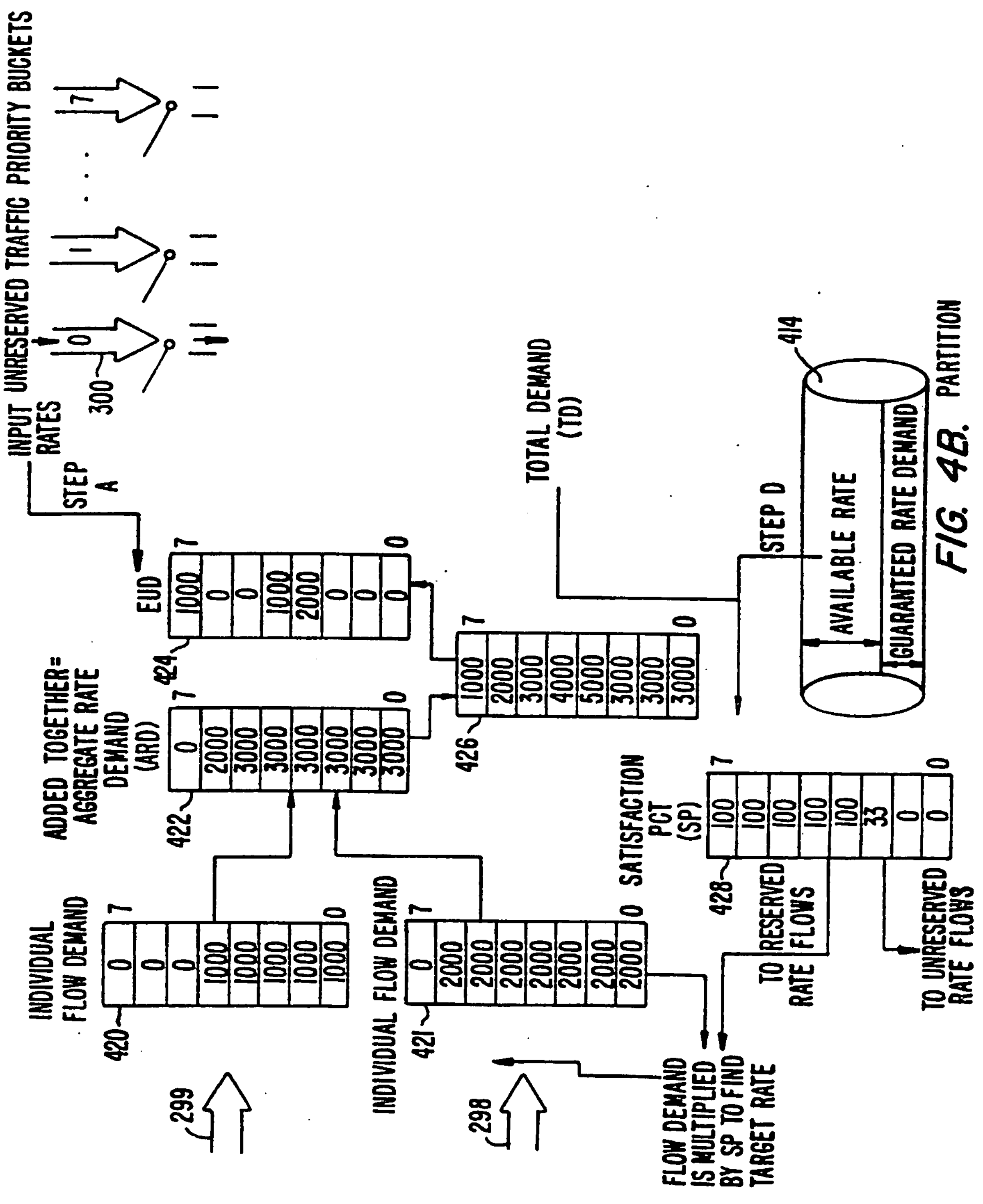
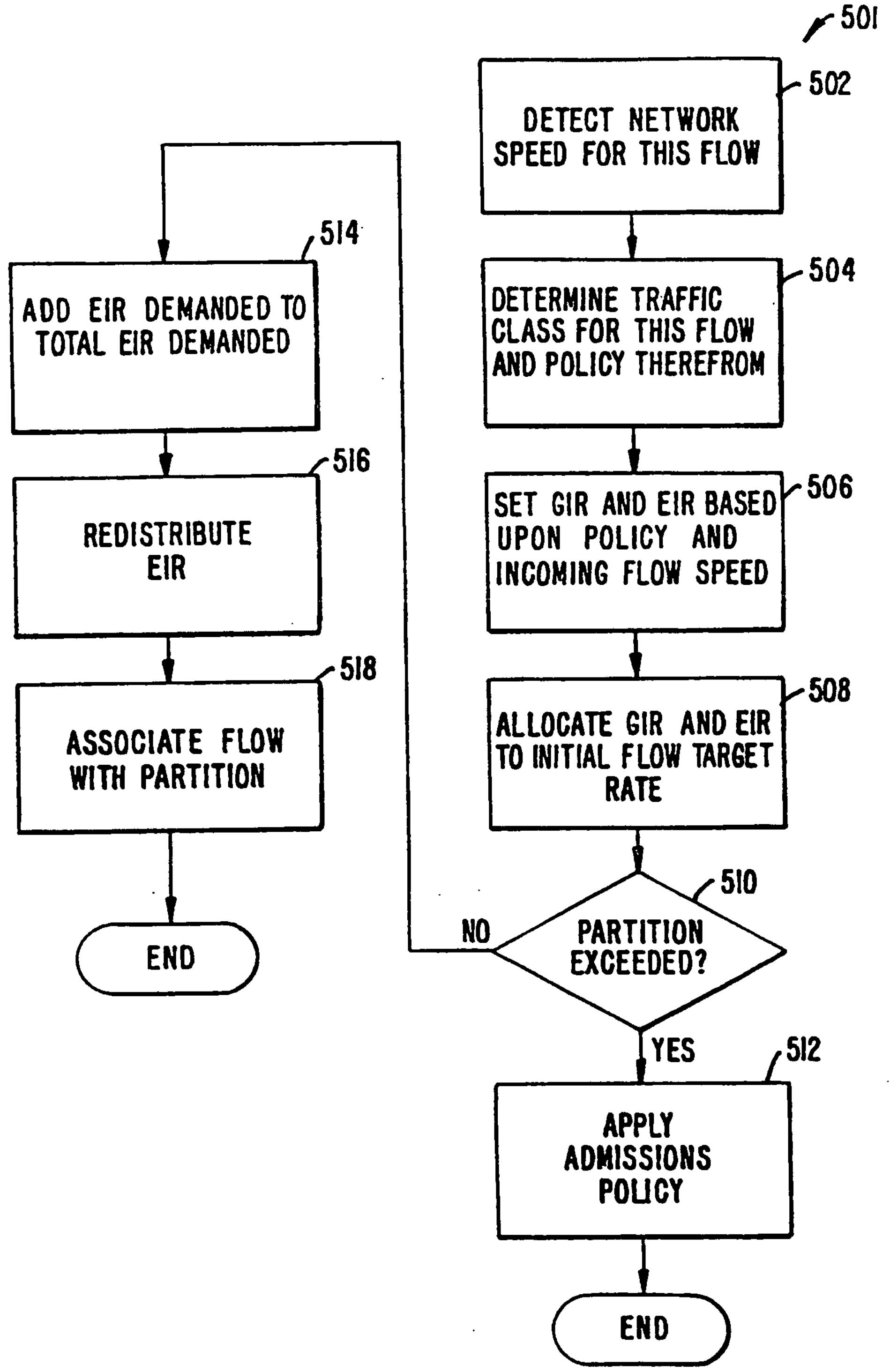

INPUT UNRESERVED TRAFFIC PRIORITY BUCKETS
RATES
300
0
1
7
STEP A
EUD
424
1000 0 0 1000 2000 0 0 0
7
0
ADDED TOGETHER= AGGREGATE RATE DEMAND (ARD)
422
0 2000 3000 3000 3000 3000 3000 3000
426
1000 2000 3000 4000 5000 3000 3000 3000
TOTAL DEMAND (TD)
STEP D
414
AVAILABLE RATE
GUARANTEED RATE DEMAND
PARTITION
INDIVIDUAL FLOW DEMAND
420
0 0 0 1000 1000 1000 1000 1000
INDIVIDUAL FLOW DEMAND
421
0 2000 2000 2000 2000 2000 2000 2000
SATISFACTION PCT (SP)
428
100 100 100 100 100 33 0 0
TO RESERVED RATE FLOWS
TO UNRESERVED RATE FLOWS
FLOW DEMAND IS MULTIPLIED BY SP TO FIND TARGET RATE
299
298

FIG. 4B.



501
DETECT NETWORK SPEED FOR THIS FLOW
502
DETERMINE TRAFFIC CLASS FOR THIS FLOW AND POLICY THEREFROM
504
SET GIR AND EIR BASED UPON POLICY AND INCOMING FLOW SPEED
506
ALLOCATE GIR AND EIR TO INITIAL FLOW TARGET RATE
508
PARTITION EXCEEDED?
510
NO
YES
APPLY ADMISSIONS POLICY
512
END
ADD EIR DEMANDED TO TOTAL EIR DEMANDED
514
REDISTRIBUTE EIR
516
ASSOCIATE FLOW WITH PARTITION
518
END

FIG. 5A.

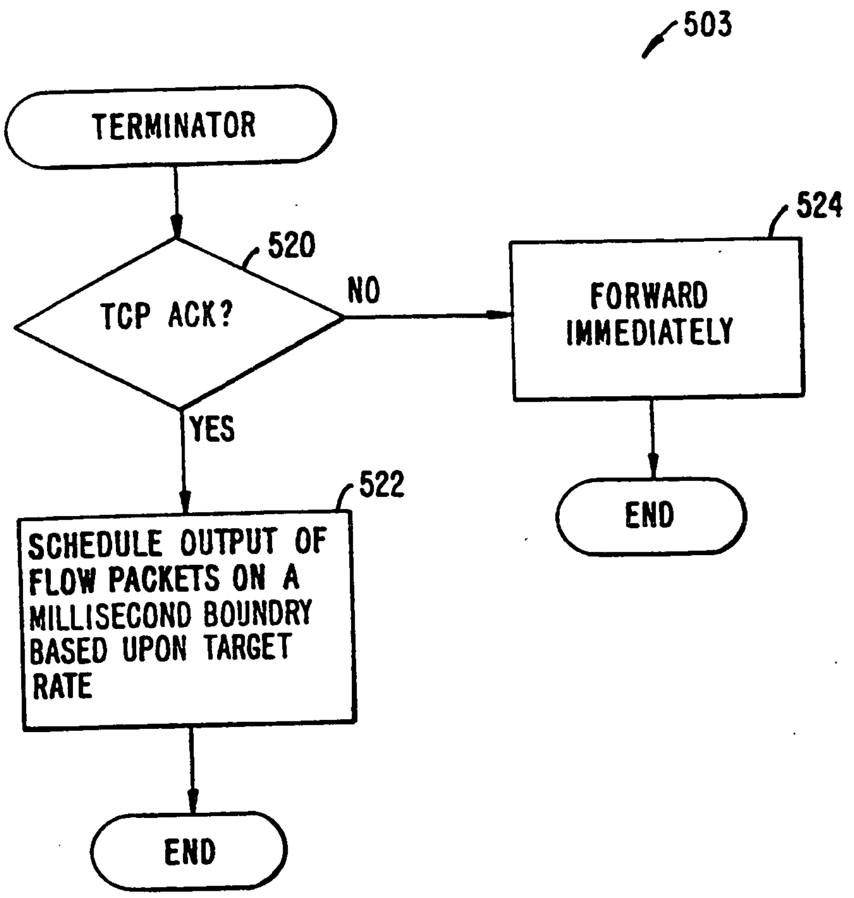

503
TERMINATOR
520
TCP ACK?
NO
YES
524
FORWARD IMMEDIATELY
END
522
SCHEDULE OUTPUT OF FLOW PACKETS ON A MILLISECOND BOUNDRY BASED UPON TARGET RATE
END

FIG. 5B.

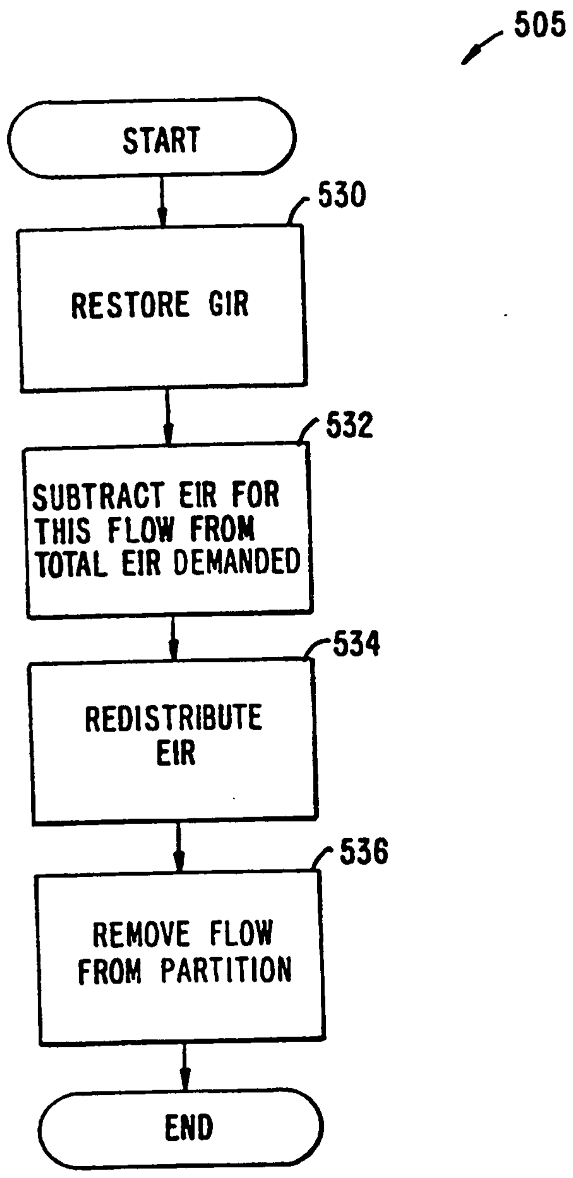

505
START
530
RESTORE GIR
532
SUBTRACT EIR FOR THIS FLOW FROM TOTAL EIR DEMANDED
534
REDISTRIBUTE EIR
536
REMOVE FLOW FROM PARTITION
END

FIG. 5C.

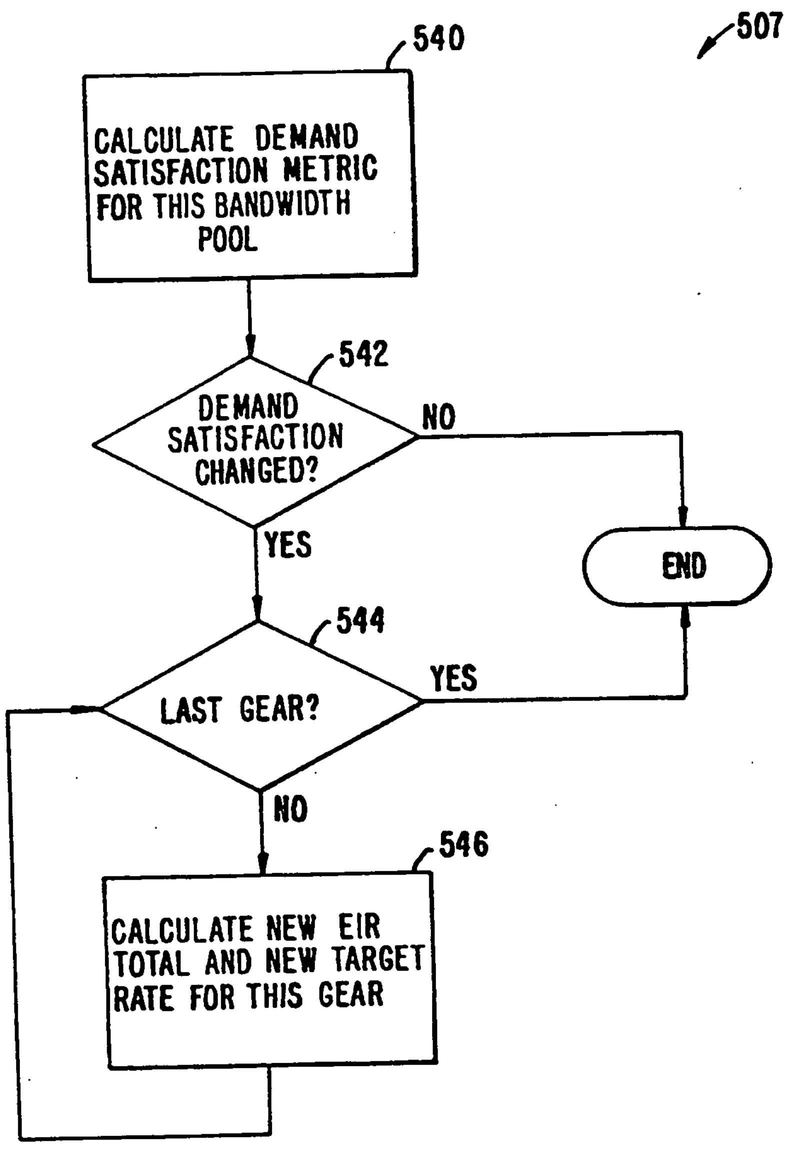

507
540
CALCULATE DEMAND SATISFACTION METRIC FOR THIS BANDWIDTH POOL
542
DEMAND SATISFACTION CHANGED?
NO
YES
END
544
LAST GEAR?
YES
NO
546
CALCULATE NEW EIR TOTAL AND NEW TARGET RATE FOR THIS GEAR

FIG. 5D.

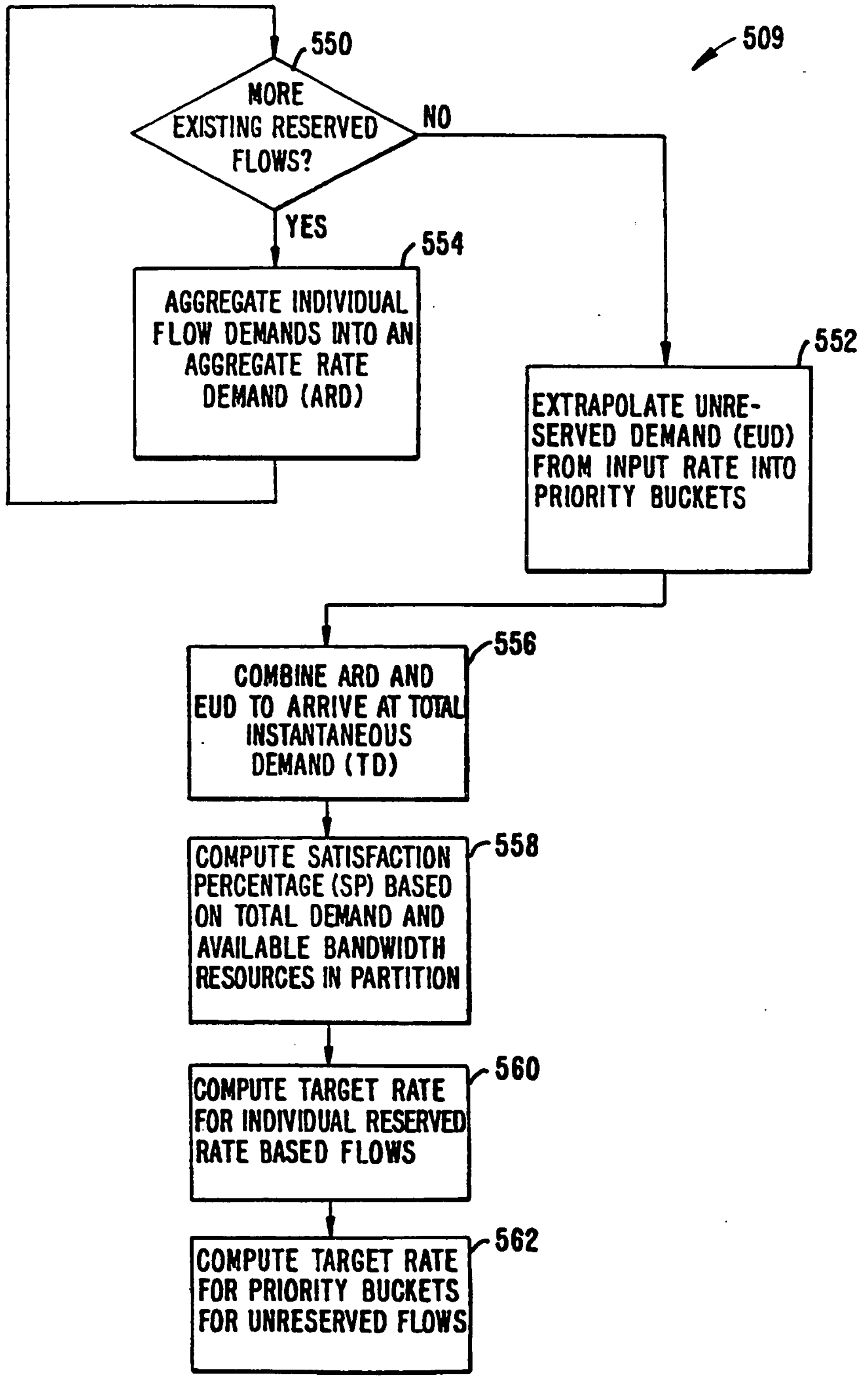

509
550
MORE EXISTING RESERVED FLOWS?
NO
YES
554
AGGREGATE INDIVIDUAL FLOW DEMANDS INTO AN AGGREGATE RATE DEMAND (ARD)
552
EXTRAPOLATE UNRESERVED DEMAND (EUD) FROM INPUT RATE INTO PRIORITY BUCKETS
556
COMBINE ARD AND EUD TO ARRIVE AT TOTAL INSTANTANEOUS DEMAND (TD)
558
COMPUTE SATISFACTION PERCENTAGE (SP) BASED ON TOTAL DEMAND AND AVAILABLE BANDWIDTH RESOURCES IN PARTITION
560
COMPUTE TARGET RATE FOR INDIVIDUAL RESERVED RATE BASED FLOWS
562
COMPUTE TARGET RATE FOR PRIORITY BUCKETS FOR UNRESERVED FLOWS

FIG. 5E.

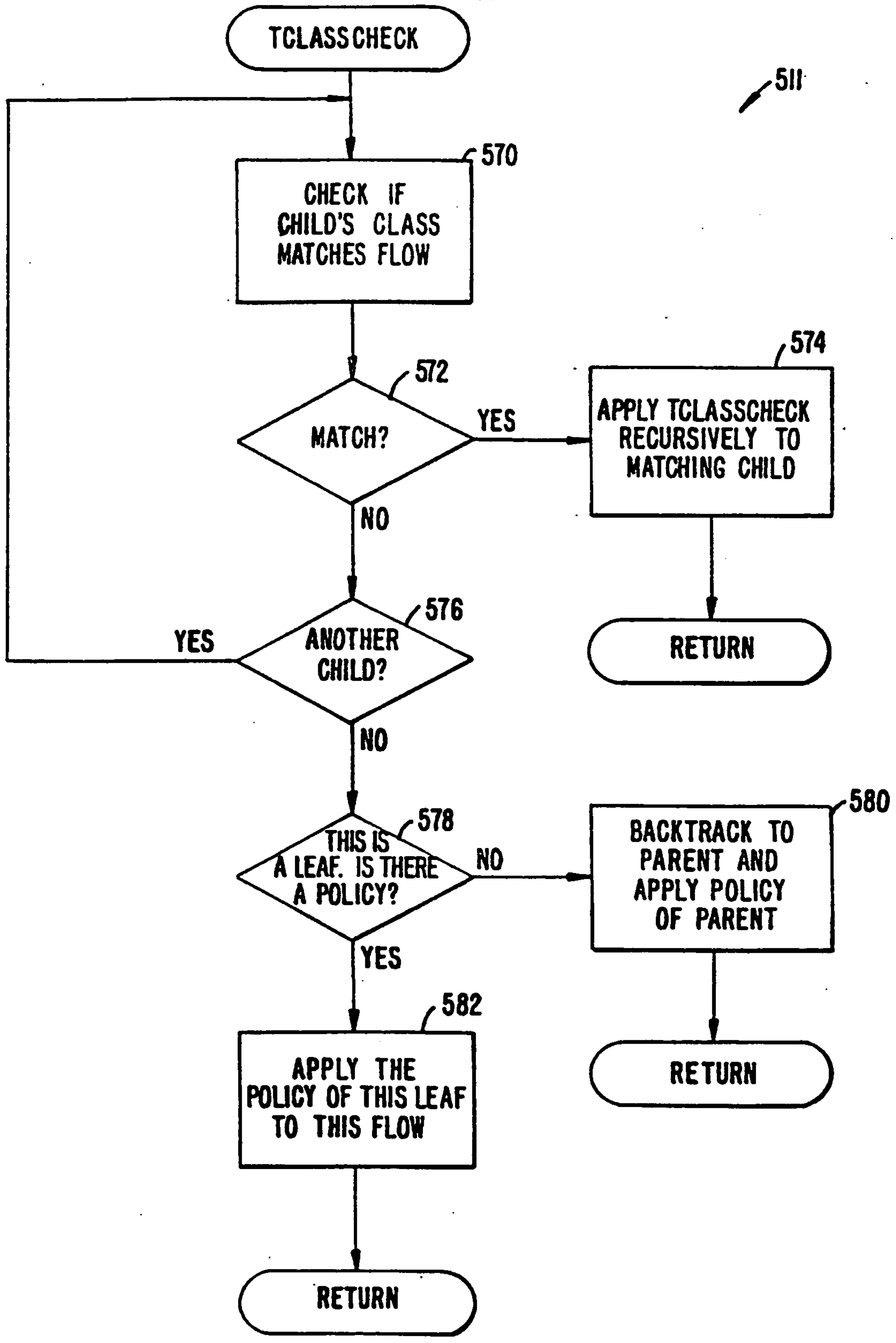

TCLASSCHECK
511
570
CHECK IF CHILD'S CLASS MATCHES FLOW
572
MATCH?
YES
574
APPLY TCLASSCHECK RECURSIVELY TO MATCHING CHILD
RETURN
NO
576
ANOTHER CHILD?
YES
NO
578
THIS IS A LEAF. IS THERE A POLICY?
NO
580
BACKTRACK TO PARENT AND APPLY POLICY OF PARENT
RETURN
YES
582
APPLY THE POLICY OF THIS LEAF TO THIS FLOW
RETURN

FIG. 5F.

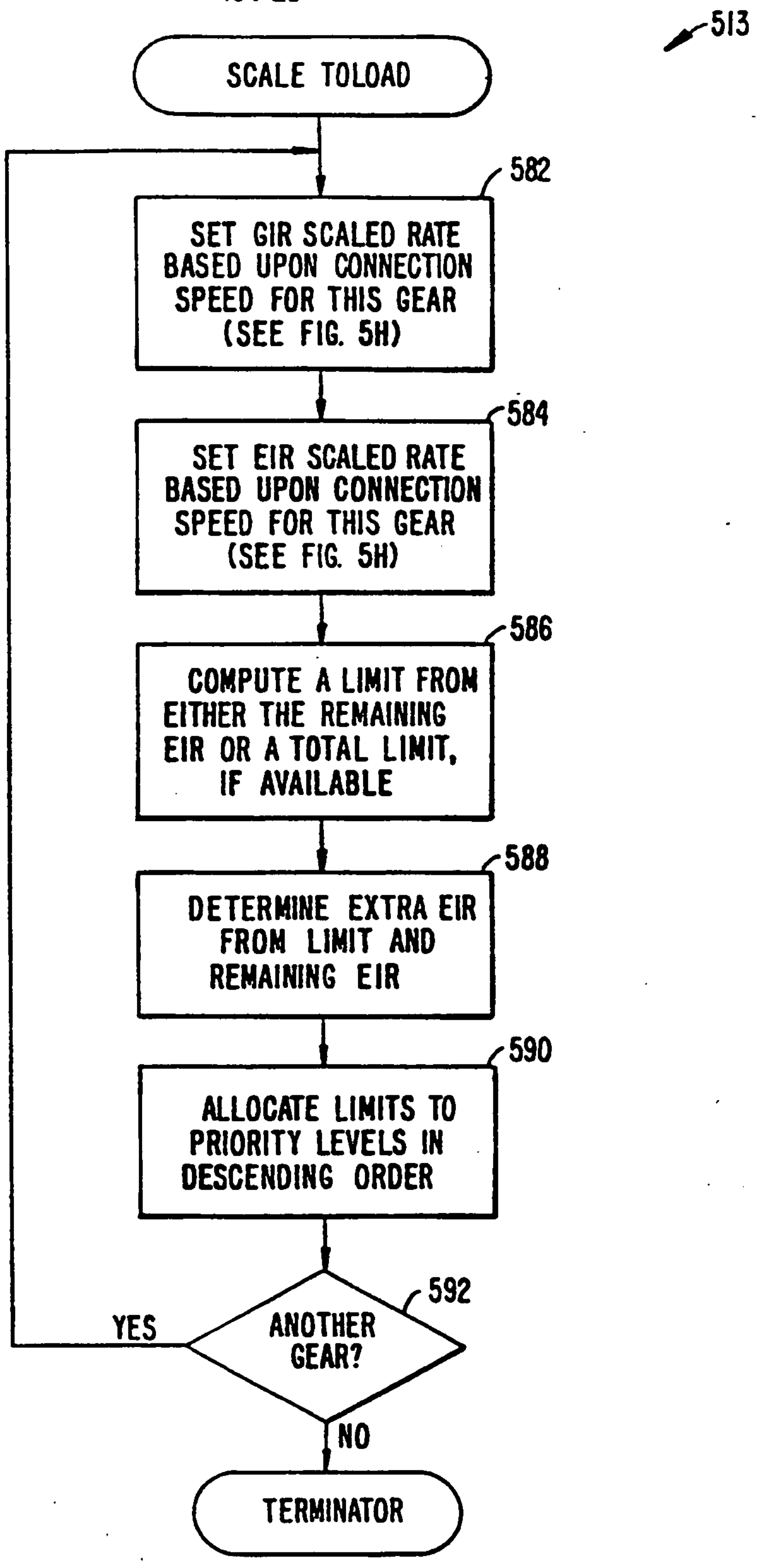

513
SCALE TOLOAD
582
SET GIR SCALED RATE BASED UPON CONNECTION SPEED FOR THIS GEAR (SEE FIG. 5H)
584
SET EIR SCALED RATE BASED UPON CONNECTION SPEED FOR THIS GEAR (SEE FIG. 5H)
586
COMPUTE A LIMIT FROM EITHER THE REMAINING EIR OR A TOTAL LIMIT, IF AVAILABLE
588
DETERMINE EXTRA EIR FROM LIMIT AND REMAINING EIR
590
ALLOCATE LIMITS TO PRIORITY LEVELS IN DESCENDING ORDER
592
ANOTHER GEAR?
YES
NO
TERMINATOR

FIG. 5G.
(STEP 506 OF FIG. 5A)

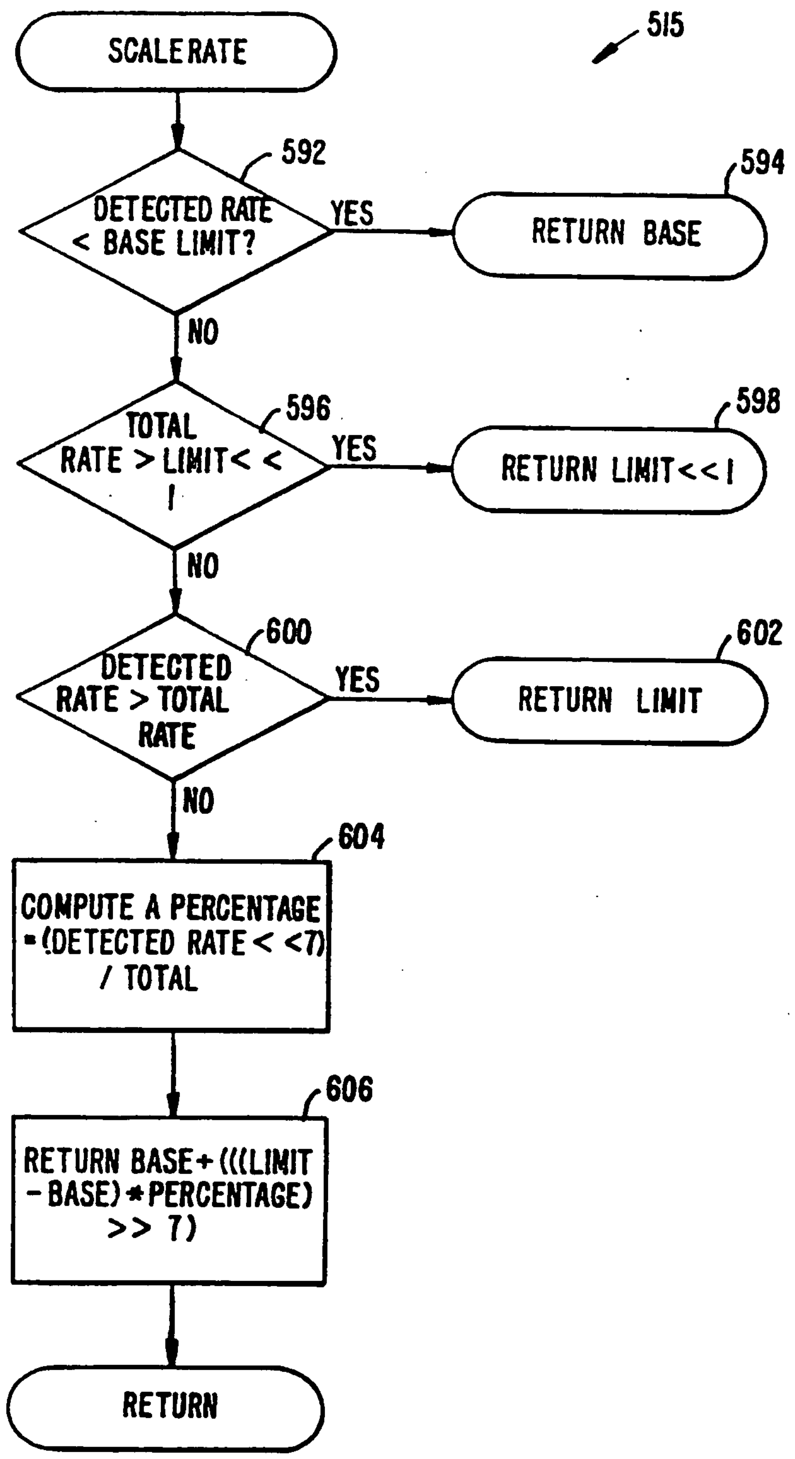

515
SCALE RATE
592
DETECTED RATE < BASE LIMIT?
YES
594
RETURN BASE
NO
596
TOTAL RATE > LIMIT< < 1
YES
598
RETURN LIMIT<<1
NO
600
DETECTED RATE > TOTAL RATE
YES
602
RETURN LIMIT
NO
604
COMPUTE A PERCENTAGE =(DETECTED RATE< <7) / TOTAL
606
RETURN BASE+(((LIMIT - BASE)*PERCENTAGE) >> 7)
RETURN

FIG. 5H.
(STEPS 582 AND 584 OF FIG. 5G)